Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 357 (2006) 100–135

Theoretical Computer Science

www.elsevier.com/locate/tcs

From truth to computability I☆

Giorgi Japaridze

Department of Computing Sciences, Villanova University, 800 Lancaster Avenue, Villanova, PA 19085, USA

Abstract

The recently initiated approach called computability logic is a formal theory of interactive computation. It understands computational problems as games played by a machine against its environment, and uses logical formalism to describe valid principles of computability, with formulas representing computational problems and logical operators standing for operations on computational problems. The concept of computability that lies under this approach is a generalization of Church–Turing computability from simple, two-step (question/answer, input/output) problems to problems of arbitrary degrees of interactivity. Restricting this concept to predicates, which are understood as computational problems of zero degree of interactivity, yields exactly classical truth. This makes computability logic a generalization and refinement of classical logic.

The foundational paper "Introduction to computability logic" [G. Japaridze, Ann. Pure Appl. Logic 123 (2003) 1–99] was focused on semantics rather than syntax, and certain axiomatizability assertions in it were only stated as conjectures. The present contribution contains a verification of one of such conjectures: a soundness and completeness proof for the deductive system **CL3** which axiomatizes the most basic first-order fragment of computability logic called the finite-depth, elementary-base fragment. **CL3** is a conservative extension of classical predicate calculus in the language which, along with all of the (appropriately generalized) logical operators of classical logic, contains propositional connectives and quantifiers representing the so called choice operations. The atoms of this language are interpreted as elementary problems, i.e. predicates in the standard sense. Among the potential application areas for **CL3** are the theory of interactive computation, constructive applied theories, knowledgebase systems, systems for resource-bound planning and action.

This paper is self-contained as it reintroduces all relevant definitions as well as main motivations. It is meant for a wide audience and does not assume that the reader has specialized knowledge in any particular subarea of logic or computer science.

MSC: primary: 03B47 secondary: 03F5003B70; 68Q10; 68T27; 68T30; 91A05

Keywords: Computability logic; Interactive computation; Game semantics; Linear logic; Constructive logics; Knowledge bases

1. Introduction

The question "What can be computed?'' is fundamental to theoretical computer science. The approach initiated recently in [10], called *computability logic*, is about answering this question in a systematic way using logical formalism, with formulas understood as computational problems and logical operators as operations on computational problems.

E-mail address: giorgi.japaridze@villanova.edu.

☆ This material is based upon work supported by the National Science Foundation under Grant No. 0208816.

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.03.014

The collection of operators used in [10] to form the language of computability logic can be seen as a non-disjoint union of those of classical, intuitionistic and—in a very generous sense—linear logics, with the computational semantics of classical operators fully consistent with their standard meaning, and the semantics of the intuitionistic-logic and "linear-logic'' operators formalizing the (so far rather abstract) computability and resource intuitions traditionally associated with those two logics. This collection captures a set of most basic and natural operations on computational problems. But it generally remains, and will apparently always remain, open to different sorts of interesting extensions, depending on particular needs and taste. Some of such extensions are outlined in [12]. Due to the fact that the language of computability logic has no clear-cut boundaries, every technical result in this area will deal with some fragment of that logic rather than the whole logic. The result presented in this paper concerns what in [10] was called the *finite-depth*, *elementary-base* fragment.

This fragment is axiomatized as a rather unusual type of a deductive system called **CL3**. It is a conservative extension of classical first-order logic in a language obtained by incorporating into the formalism of the latter the "additive'' and "multiplicative'' groups of to what we—with strong reservations—referred as "linear-logic operators''. The main technical result of this paper is a proof of soundness and completeness for **CL3** with respect to computability semantics. A secondary result is a proof of decidability for the classical-quantifier-free (yet first-order) fragment of **CL3**. These proofs are given in Part 2. Part 1 is mainly devoted to a relatively brief (re)introduction to the relevant fragment and relevant aspects of computability logic, serving the purpose of keeping the paper self-contained both technically and motivationally. A more detailed and fundamental introduction to computability logic can be found in [10]. A considerably less technical and more compact—yet comprehensive and self-contained—overview of computability logic is given in [12], reading which is most recommended for the first acquaintance with the subject and for a better appreciation of the import of the present results. The soundness and completeness of the propositional fragment **CL1** of **CL3** has been proven in [11].

Traditionally construed computational problems correspond to interfaces in transformational programs where the interaction between a system and its environment is simple and consists of two steps: querying the system and generating an answer. The computational problems that our approach deals with are more general in that the underlying interfaces may have arbitrary complexity. Such problems and the corresponding computations can be called *interactive* as they model potentially long dialogues between the system and the environment. From the technical point of view, computability logic is a game logic, because it defines interactive computations as games. There is an extensive literature on "game-style'' models of computation in theoretical computer science: alternating Turing machines, interactive proof systems, etc., that are typically only interesting in the context of computational complexity. Our approach, which is concerned with computability rather than complexity and deals with deterministic rather than non-deterministic choices, at present is only remotely related to that line of research, and the similarity is more terminological than conceptual. From the other, "games in logic'' or "game semantics for linear logic'' line, the closest to our present study appears to be Blass's work [3], and less so some later studies by Abramsky and Jagadeesan [1], Hyland and Ong [6] and others. See [10] for discussions of how other approaches compare with ours.

There are considerable overlaps between the motivations and philosophies of linear (as well as intuitionistic) and computability logics, based on which [10] employed some linear-logic terminology. It should be pointed out, however, that computability logic is by no means *about* linear logic. Unlike most of the other game semantics approaches, it is *not* an attempt to use games to construct good models for Girard's linear logic, Heyting's intuitionistic calculus or any other, already given popular syntactic targets. Rather, computability logic evolves by the more and only natural scheme "from semantics to syntax'': it views games as foundational entities in their own right, and then explores the logical principles validated by them. Its semantics, in turn, follows the scheme "from truth to computability''. It starts from the classical concept of truth and generalizes it to the more constructive, meaningful and useful concept of computability. As we are going to see, classical truth is nothing but a special case of computability; correspondingly, classical logic is nothing but a special fragment of computability logic and of **CL3** in particular.

The scope of the significance of our study is not limited to logic or theory of computing. As we will see later and more convincingly demonstrated in [10] and [12], some other application areas include constructive applied theories, knowledgebase systems, or resource-bound systems for planning and action.

Part 1

This part briefly reintroduces the subject and states the main results of the paper.

2. Computational problems

The concept of computability on which the semantics of our logic is based is a natural but non-trivial generalization of Church–Turing computability from simple, two-step (question/answer, or input/output) problems to problems of arbitrary degrees and forms of interactivity where, in the course of interaction between the machine and the environment, input and output can be multiple and interlaced, perhaps taking place throughout the entire process of computation rather than just at the beginning (input) and the end (output) as this is the case with simple problems. Technically the concept is defined in terms of games: an interactive computational problem/task is a game between a machine and the environment, where dynamic input steps are called environment's moves, and output steps called machine's moves.

The necessity in having a clear mathematical model of interactive computation hardly requires any justification: after all, most tasks that real computers and computer networks perform are truly interactive. And this sort of tasks cannot always be reduced to simple series of (the well-studied and well-modeled) two-step tasks. For example, interactive tasks involving multiple concurrent subtasks naturally generate situations/positions where both parties may have meaningful actions to take, and it may be up to the player whether to try to make a move or wait to see how things evolve, perhaps performing some vital computations while waiting and watching. [1] It is unclear whether the steps corresponding to such situations should be labeled as 'machine-to-move' or 'environment-to-move', which makes it impossible to break the whole process into consecutive pairs or alternately labeled steps.

Standard game-semantical approaches that understand players' strategies as functions from positions to moves [2] fail to capture the substance of interaction in full generality, as they essentially try to reduce interactive processes to simple chains of question/answer events. This is so because the 'strategy = function' approach inherently only works when at every step of the play the player who is expected to make a move is uniquely determined. Let us call this sort of games *strict*. Strictness is typically achieved by having what in [2] is called *procedural rules* or equivalent—rules strictly regulating who and when should move, the most standard procedural rule being the one according to which players should take turns in alternating order.

One of the main novel and distinguishing features of our games among the other concepts of games studied in the logical literature—including the one tackled by the author [7] earlier—is the absence of procedural rules, based on which our games can be called *free*. In these games, either player is free to make any move at any time. Instead of having procedural rules common for all games, each particular game comes with its own what we call *structural rules*. These are rules telling what moves are *legal* for a given player in a given position. Making an illegal move by a player is possible but it results in a loss for that player. The difference between procedural and structural rules is not just terminological. Unlike the standard procedural rules that allow only one player to move (at least move without penalty) in every given situation, structural rules can be arbitrarily lenient. In particular, they do not necessarily have to satisfy the condition that in every position at most one of the players may have legal moves. When, however, this condition still *is* satisfied, we essentially get the above-mentioned strict type of a game: the structural rules of such a game can be thought of as procedural rules according to which the player that is expected to move in a given non-terminal position is the (now uniquely determined) one that has legal moves in that position. Strict games are thus special cases of our more general free games. The latters present a more adequate and apparently universal modeling tool for interactive tasks.

Strategies for playing free games can and should no longer be defined as functions from positions to moves. In the next section we will define them as higher-level entities called play machines.

To define our games formally, we fix several classes of objects and dedicated (meta-) variables for them. By placing the common name for objects between braces we denote the set of all objects of that type. Say, {*variables*} stands for the set of all variables. These objects are:

- *Variables*, with $\{variables\} = \{v_0, v_1, v_2, \ldots\}$. Letters x, y, z will be used as metavariables for variables.
- *Constants*, with $\{constants\} = \{0, 1, 2, \ldots\}$. Letter c will be used as a metavariable for constants.
- *Terms*, with $\{terms\} = \{variables\} \cup \{constants\}$. Letter t will be used as a metavariable for terms.
- *Valuations*, defined as any functions of the type $\{variables\} \to \{constants\}$. A metavariable for valuations: e. Each valuation e extends to a function of the type $\{terms\} \to \{constants\}$ by stipulating that, for every constant c, $e(c) = c$.

[1] See Sections 3 and 15 of [10] for more detailed discussions and examples.

[2] Often some additional restrictions are imposed on this sort of strategies. Say, in Abramsky's tradition, strategies only look at the other player's immediately preceeding moves.

- *Moves*, defined as any finite strings over the standard keyboard alphabet plus the symbol ♠. Metavariables for moves: α, β, γ.
- *Players*, with $\{players\} = \{\top, \bot\}$. Here and from now on $\top$ and $\bot$ are symbolic names for the machine and the environment, respectively. Letter $\wp$ will range over players, with $\neg\wp$ meaning "$\wp$'s adversary'', i.e. the player that is not $\wp$.
- *Labeled moves*, or *labmoves*. They are defined as any moves prefixed with $\top$ or $\bot$, with such a prefix (*label*) indicating who has made the move.
- *Runs*, defined as any (finite or infinite) sequences of labmoves. Metavariables for runs: Γ, Δ.
- *Positions*, defined as any finite runs. A metavariable for positions: Φ.

Runs and positions will often be delimited with "⟨'' and "⟩'', with $\langle\rangle$ thus denoting the *empty run*. The meaning of an expression such as $\langle\Phi, \wp\alpha, \Gamma\rangle$ must be clear: this is the result of appending to the position Φ the position $\langle\wp\alpha\rangle$ and then the run Γ.

Definition 2.1. A *game* is a pair $A = (\mathbf{Lr}^A, \mathbf{Wn}^A)$, where:

- $\mathbf{Lr}^A$ is a function that sends each valuation e to a set $\mathbf{Lr}^A_e$ of runs, such that the following two conditions are satisfied:
  (a) A run is in $\mathbf{Lr}^A_e$ iff all of its non-empty finite initial segments are in $\mathbf{Lr}^A_e$.
  (b) No run containing the (whatever-labeled) move ♠ is in $\mathbf{Lr}^A_e$.
  Elements of $\mathbf{Lr}^A_e$ are called *legal runs* of A with respect to e, and all other runs called *illegal*. In particular, if the last move of the shortest illegal initial segment of Γ is $\wp$-labeled, then Γ is said to be a *$\wp$-illegal run* of A with respect to e.
- $\mathbf{Wn}^A$ is a function of the type $\{valuations\} \times \{runs\} \to \{players\}$ such that, writing $\mathbf{Wn}^A_e\langle\Gamma\rangle$ for $\mathbf{Wn}^A(e, \Gamma)$, the following condition is satisfied:
  (c) If Γ is a $\wp$-illegal run of A with respect to e, then $\mathbf{Wn}^A_e\langle\Gamma\rangle = \neg\wp$.

To what we earlier referred as "structural rules'' are thus represented by the **Lr** component of a game, and we call it the *structure* of that game. The meaning of the **Wn** component, called the *content*, is that it tells us who has won a given run of the game. When $\mathbf{Wn}^A_e\langle\Gamma\rangle = \wp$, we say that Γ is a *$\wp$-won* (or *won by* $\wp$) run of A with respect to e.

Just as predicates (their truth values) in classical logic generally depend on how certain variables are interpreted, so do games: both the structure and the content of a game take valuation e as a parameter. We will typically omit this parameter when it is irrelevant or clear from the context.

Meaning by an *illegal move* a move adding which (with the corresponding label) to the given position makes it illegal, condition (a) of Definition 2.1 corresponds to the intuition that a run is legal iff no illegal moves have been made in it. This automatically makes the empty run $\langle\rangle$ a legal run of every game. Our selection of the set of moves is very generous, and it is natural and technically very convenient to assume that certain moves will never be legal. According to condition (b), ♠ is such a move. As for condition (c), it tells us that an illegal run is always lost by the player who has made the first illegal move.

We say that a game A *depends on* a variable x iff there are two valuations e_1 and e_2 that agree on all variables except x such that either $\mathbf{Lr}^A_{e_1} \neq \mathbf{Lr}^A_{e_2}$ or $\mathbf{Wn}^A_{e_1} \neq \mathbf{Wn}^A_{e_2}$.

A game A is said to be *finitary* iff there is a finite set $\vec{x}$ of variables such that, for any two valuations e_1 and e_2 that agree on all variables from $\vec{x}$, we have $\mathbf{Lr}^A_{e_1} = \mathbf{Lr}^A_{e_2}$ and $\mathbf{Wn}^A_{e_1} = \mathbf{Wn}^A_{e_2}$. Otherwise A is *infinitary*. One can easily show that for each finitary game A there is a unique smallest set $\vec{x}$ that satisfies the above condition—in particular, the elements of this $\vec{x}$ are exactly the variables on which A depends. A finitary game that depends on exactly n variables is said to be *n-ary*.

A *constant game* means a 0-ary game. There is a natural operation, called *instantiation*, of the type $\{valuations\} \times \{games\} \to \{constant\ games\}$. The result of applying this operation to (e, A) is denoted $e[A]$. Intuitively, $e[A]$ is the constant game obtained from A by fixing the values of all variables to the constants assigned to them by e. Formally, game $e[A]$ is defined by stipulating that, for any valuation e', $\mathbf{Lr}^{e[A]}_{e'} = \mathbf{Lr}^A_e$ and $\mathbf{Wn}^{e[A]}_{e'} = \mathbf{Wn}^A_e$. This makes e' irrelevant, so that it can be omitted and we can just write $\mathbf{Lr}^{e[A]}$ and $\mathbf{Wn}^{e[A]}$. For any game A, these two expressions mean the same as $\mathbf{Lr}^A_e$ and $\mathbf{Wn}^A_e$.

Games whose **Lr** component does not depend on the valuation parameter are said to be *unistructural*. Constant games are thus special cases of unistructural games where the **Wn** component does not depend on valuation, either. Formally, a game A is unistructural iff, for any two valuations e_1 and e_2, $\mathbf{Lr}^A_{e_1} = \mathbf{Lr}^A_{e_2}$.

We say that a game A is *finite-depth* iff there is a (smallest) integer n, called the *depth of* A, such that, for every valuation e and run Γ with $\Gamma \in \mathbf{Lr}_e^A$, the length of Γ does not exceed n. Games of depth 0 are said to be *elementary*. Thus, elementary games are games that have no legal moves: the empty run $\langle\rangle$ is the only legal run of such games. This automatically makes all elementary games unistructural.

Constant elementary games are said to be *trivial*. Obviously there are exactly two trivial games. We denote them by the same symbols $\top$ and $\bot$ as we use for the two players. In particular, $\top$ is the (unique) trivial game with $\mathbf{Wn}^{\top}\langle\rangle = \top$, and $\bot$ is the (unique) trivial game with $\mathbf{Wn}^{\bot}\langle\rangle = \bot$.

Let us agree to understand classical *predicates*—in particular, predicates on $\{0, 1, 2, \ldots\}$—as functions from valuations to $\{\top, \bot\}$ rather than, as more commonly done, functions from tuples of constants to $\{\top, \bot\}$. Say, $x > y$ is the predicate that is true at a valuation e (returns $\top$ for it) iff $e(x) > e(y)$. This understanding of predicates is technically more convenient, and is also slightly more general as it captures infinite-arity predicates along with finite-arity ones.

All elementary games have the same structure, so their trivial **Lr** component can be ignored and each such game identified with its **Wn** component; furthermore, by setting the run parameter to its only relevant value $\langle\rangle$ in such games, **Wn** becomes a function of the type $\{\textit{valuations}\} \rightarrow \{\textit{players}\}$, i.e. exactly what we call a predicate. We thus get a natural one-to-one correspondence between elementary games and predicates: every predicate p can be thought of as the (unique) elementary game A such that $\mathbf{Wn}_e^A\langle\rangle = \top$ iff p is true at e; and vice versa: every elementary game A thought of as the predicate p that is true at e iff $\mathbf{Wn}_e^A\langle\rangle = \top$. With this correspondence in mind, we will be using the terms "predicate'' and "elementary game'' as synonyms. So, computability logic understands each predicate p as a game of zero degree of interactivity, (the only legal run $\langle\rangle$ of) which is automatically won by the machine if p is true, and lost if p is false. This makes the classical concept of predicates a special case of our concept of computational problems; correspondingly, the classical concept of truth is going to be a special case of our concept of computability—in particular, computability restricted to elementary games.

The class of games in the above-defined sense is general enough to model anything that we would call a (two-player, two-outcome) interactive problem. However, it is too general. There are games where the chances of a player to succeed essentially depend on the relative speed at which its adversary responds and, as it is not clear what particular speed would be natural to assume for the environment, we do not want to consider that sort of games meaningful computational problems. A simple example would be the game where all non-♠ moves are legal and that is won by the player who moves first. This is merely a contest of speed.

Below we define a subclass of games called static. Intuitively, static games are games where speed is irrelevant: in order to succeed, only matters *what* to do (strategy) rather than *how fast* to do (speed).

We say that a run Δ is a $\wp$-*delay* of a run Γ iff the following two conditions are satisfied:

- for each player $\wp'$, the subsequence of the $\wp'$-labeled moves of Δ is the same as that of Γ, and
- for any $n, k \geqslant 1$, if the nth $\wp$-labeled move is made later than (is to the right of) the kth $\neg\wp$-labeled move in Γ, then so is it in Δ.

The above means that in Δ each player has made the same sequence of moves as in Γ, only, in Δ, $\wp$ might have been acting with some delay.

Definition 2.2. A game A is said to be *static* iff, whenever $\mathbf{Wn}_e^A\langle\Gamma\rangle = \wp$ and Δ is a $\wp$-delay of Γ, we have $\mathbf{Wn}_e^A\langle\Delta\rangle = \wp$.

Roughly, in a static game, if a player can succeed when acting fast, it will remain equally successful acting the same way but slowly. This releases the player from any pressure for time and allows it to select its own pace for the game. The following fact is a rather straightforward observation:

Proposition 2.3. *All elementary games are static.*

One of the main theses on which computability logic relies philosophically is that the concept of static games is an adequate formal counterpart of our intuitive notion of "pure'', speed-independent computational problems. See Section 4 of [10] for a detailed discussion and examples in support of this thesis.

Now we are ready to formally clarify what we mean by computational problems: we use the term "(interactive) *computational problem*'' (or simply "*problem*'') as a synonym of "static game''.

As shown in [10] (Proposition 4.8), all strict games are static. But not vice versa. The class of static games is substantially more general, and is free of the limitations of strict games discussed earlier. The closure of the set of all

predicates under the game operations that we will define in Section 4 forms a natural class of static yet free games. Section 3 of [10] shows an example of a natural problem from this class that is impossible to model with strict games.

3. Computability

The definitions that we give in this section are semiformal and incomplete. All of the omitted technical details are however rather standard and can be easily restored by anyone familiar with Turing machines. If necessary, the corresponding detailed definitions can be found in Part II of [10].

The central point of our philosophy is to require that agent $\top$ be implementable as a computer program, with effective and fully determined behavior. On the other hand, the behavior (including speed) of agent $\bot$, who represents a capricious user or the blind forces of nature, can be arbitrary. This intuition is captured by the model of interactive computation where $\top$ is formalized as what we call HPM.

A *hard-play machine (HPM)* $\mathcal{H}$ is a Turing machine with the capability of making moves. At any time, the current position of the game is fully visible to this machine, as well as it is fully informed about the valuation with respect to which the outcome of the play will be evaluated. This effect is achieved by letting the machine have—along with the ordinary read/write *work tape*—two additional read-only tapes: the *valuation tape* and the *run tape*. The former spells some valuation e by listing constants in the lexicographic order of the corresponding variables. Its contents remain unchanged throughout the work of the machine. As for the run tape, it serves as a dynamic input, spelling, at any time, the current position of the game. Every time one of the players makes a move, that move (with the corresponding label) is automatically appended to the contents of the run tape.

As always, the computation proceeds in discrete steps, also called *clock cycles*. The technical details about how exactly $\mathcal{H}$ makes a move α are not very interesting, but for clarity let us say that this is done by constructing α in a certain section (say, the beginning) of the work tape and then entering one of the specially designated states called *move states*. Thus, $\mathcal{H}$ can make at most one move per clock cycle. On the other hand, as we noted, there are no limitations to the relative speed of the environment, so the latter can make any finite number of moves per cycle. We assume that the run tape remains stable during a clock cycle and is updated only on a transition from one cycle to another. Again, there is flexibility in arranging details regarding what happens if both of the players make moves "simultaneously''. For clarity, we assume that if, during a given cycle, $\mathcal{H}$ makes a move α and the environment makes moves $\beta_1, \ldots, \beta_n$, then the position spelled on the run tape throughout the next cycle will be the result of appending $\langle \bot\beta_1, \ldots, \bot\beta_n, \top\alpha\rangle$ to the current position.

A *configuration* of $\mathcal{H}$ is defined in the standard way: this is a full description of the ("current'') state of the machine, the locations of its three scanning heads, and the contents of its tapes, with the exception that, in order to make finite descriptions of configurations possible, we do not formally include a description of the unchanging contents of the valuation tape as a part of configuration, but rather account for it in our definition of computation branches as this will be seen shortly. The *initial configuration* is the configuration where $\mathcal{H}$ is in its initial state and the work and run tapes are empty. A configuration C' is said to be an *e-successor* of a configuration C iff, when valuation e is spelled on the valuation tape, C' can legally follow C in the standard (standard for multitape Turing machines) sense, based on the transition function of the machine and accounting for the possibility of the above-described non-deterministic updates of the contents of the run tape. An *e-computation branch* of $\mathcal{H}$ is a sequence of configurations of $\mathcal{H}$ where the first configuration is the initial configuration and every other configuration is an e-successor of the previous one. Thus, the set of all e-computation branches captures all possible scenarios corresponding to different behaviors by $\bot$.

Each e-computation branch B of $\mathcal{H}$ incrementally spells (in the obvious sense) some run Γ on the run tape, which we call the *run spelled by* B. Then, for a game A, we write $\mathcal{H} \vDash_e A$ ("$\mathcal{H}$ *wins* A *on* e'') iff, whenever B is an e-computation branch of $\mathcal{H}$ and Γ the run spelled by B, we have $\mathbf{Wn}_e^A\langle\Gamma\rangle = \top$. And we write $\mathcal{H} \vDash A$ iff $\mathcal{H} \vDash_e A$ for every valuation e. The meaning of $\mathcal{H} \vDash A$ is that $\mathcal{H}$ *wins* (*computes*, *solves*) A. Finally, we write $\vDash A$ and say that A is *winnable* (*computable*, *solvable*) iff there is an HPM $\mathcal{H}$ with $\mathcal{H} \vDash A$.

The above "hard-play'' model of interactive computation seemingly strongly favors the environment in that the latter may be arbitrarily faster than the machine. What happens if we start limiting the speed of the environment? The answer is: *nothing* as far as computational problems are concerned. The model of computation called EPM takes the idea of limiting the speed of the environment to the extreme, yet it yields the same class of computable problems.

An *easy-play machine (EPM)* is defined in the same way as an HPM, with the only difference that now the environment can (but is not obligated to) make a move only when the machine explicitly allows it to do so, the event that we

call *granting permission*. Technically permission is granted by entering one of the specially designated states called *permission states*. The only requirement that the machine is expected to satisfy is that, as long as the adversary plays legal, the machine should grant permission every once in a while; how long that "while'' lasts, however, is totally up to the machine. This amounts to having full control over the speed of the adversary.

The above intuition is formalized as follows. We say that an e-computation branch B of a given EPM is *fair* if permission is granted infinitely many times in B. A *fair EPM* is an EPM whose every e-computation branch (for every valuation e) is fair. For an EPM $\mathcal{E}$ and valuation e, we write $\mathcal{E} \models_e A$ ("$\mathcal{E}$ *wins* A *on* e'') iff, whenever B is an e-computation branch of $\mathcal{E}$ and Γ the run spelled by B, we have:

- $\mathbf{Wn}_e^A\langle\Gamma\rangle = \top$, and
- B is fair unless Γ is a $\bot$-illegal run of A with respect to e.

Just as with HPMs, for an EPM $\mathcal{E}$, $\mathcal{E} \models A$ ("$\mathcal{E}$ *wins* (*computes*, *solves*) A'') means that $\mathcal{E} \models_e A$ for every valuation e. Note that when we deal with fair EPMs, the second one of the above two conditions is always satisfied, and then the definition of $\models_e$ is literally the same as in the case of HPMs.

Remark 3.1. When trying to show that a given EPM wins a given game, it is always perfectly safe to assume that the environment never makes an illegal move, for if it does, the machine automatically wins (unless the machine itself has made an illegal move earlier, in which case it does not matter what the environment did afterwards anyway, so that we may still assume that the environment did not make any illegal moves). Making such an assumption can often significantly simplify computability proofs.

The following fact, proven in [10, Theorem 17.2], establishes equivalence between the two models for computational problems:

Proposition 3.2. *For any static game* A, *the following statements are equivalent*:

(i) *there is an EPM that wins* A;
(ii) *there is an HPM that wins* A;
(iii) *there is a fair EPM that wins* A.

Moreover, there is an effective procedure that converts any EPM (*resp. HPM*) $\mathcal{M}$ *into an HPM* (*resp. fair EPM*) $\mathcal{N}$ *such that, for every static game* A *and valuation* e, $\mathcal{N} \models_e A$ *whenever* $\mathcal{M} \models_e A$.

The philosophical significance of this proposition is that it reinforces the thesis according to which static games are games that allow us to make full abstraction from speed. Its technical importance is related to the fact that the EPM-model is much more convenient when it comes to describing strategies as we will have a chance to see in Part 2, and is a more direct and practical formal counterpart of our intuitive notion of what could be called *interactive algorithm*.

The two models act as natural complements to each other: we can meaningfully talk about the (uniquely determined) play between a given HPM and a given EPM, while this is impossible if both players are HPMs or both are EPMs. This fact will be essentially exploited in our completeness proof for logic **CL3**, where we describe an environment's strategy as an EPM and show that no HPM can win the given game against such an EPM.

Let us agree on the following notation and terminology:

- For a run Γ, $\neg\Gamma$ denotes the result of reversing all labels in Γ, i.e. changing each labmove $\wp\alpha$ to $\neg\wp\alpha$.
- For a run Γ and a computation branch B of an HPM or EPM, we say that B *cospells* Γ iff B spells $\neg\Gamma$.

Intuitively, when a given machine $\mathcal{M}$ plays as $\bot$ (rather than $\top$), then the run that is generated by a given computation branch B of $\mathcal{M}$ is the run cospelled (rather than spelled) by B, for the moves that $\mathcal{M}$ makes get the label $\bot$, and the moves that its adversary makes get the label $\top$.

The following lemma will be used in our completeness proof for **CL3**. Its second clause assumes some standard encoding for play machines and their configurations.

Lemma 3.3. *Let* $\mathcal{E}$ *be a fair* EPM.

(a) *For any HPM* $\mathcal{H}$ *and any valuation* e, *there are a uniquely defined* e*-computation branch* $B_\mathcal{E}$ *of* $\mathcal{E}$ *and a uniquely defined* e*-computation branch* $B_\mathcal{H}$ *of* $\mathcal{H}$*—which we, respectively, call* the $(\mathcal{E}, e, \mathcal{H})$-branch *and* the $(\mathcal{H}, e, \mathcal{E})$-branch*—such that the run spelled by* $B_\mathcal{H}$ *is the run cospelled by* $B_\mathcal{E}$.

(b) *Suppose $e_0, e_1, e_2, \ldots$ are valuations such that the function g defined by $g(c, i) = e_c(v_i)$ is effective. Then there is an effective function which takes (the code of) an arbitrary HPM $\mathcal{H}$ and arbitrary non-negative integers c, n, and returns the (code of the) n'th configuration of the $(\mathcal{E}, e_c, \mathcal{H})$-branch. Similarly for the $(\mathcal{H}, e_c, \mathcal{E})$-branch.*

When $e, \mathcal{H}, \mathcal{E}, B_{\mathcal{H}}$ are as in clause (a) of the above lemma, we call the run Γ spelled by $B_{\mathcal{H}}$ *the $\mathcal{H}$ vs. $\mathcal{E}$ run on e*; then, if A is a game with $\mathbf{Wn}_e^A\langle\Gamma\rangle = \top$ (resp. $\mathbf{Wn}_e^A\langle\Gamma\rangle = \bot$), we say that $\mathcal{H}$ *wins* (resp. *loses*) A *against* $\mathcal{E}$ *on* e.

A formal proof of Lemma 3.3 is given in [10, Lemma 20.4], [3] and we will not reproduce it here. Instead, the following intuitive explanation would suffice:

Assume $\mathcal{E}$ is a fair EPM, $\mathcal{H}$ is an arbitrary HPM and e an arbitrary valuation. The play that we are going to describe is the unique play generated when the two machines play against each other, with $\mathcal{H}$ in the role of $\top$, $\mathcal{E}$ in the role of $\bot$, and valuation e spelled on the valuation tapes of both machines. We can visualize this play as follows. Most of the time during the play $\mathcal{H}$ remains inactive (sleeping); it is woken up only when $\mathcal{E}$ enters a permission state, on which event $\mathcal{H}$ makes a (one single) transition to its next computation step—that may or may not result in making a move—and goes back to sleep that will continue until $\mathcal{E}$ enters a permission state again, and so on. From $\mathcal{E}$'s perspective, $\mathcal{H}$ acts as a patient adversary who makes one or zero move only when granted permission, just as the EPM-model assumes. And from $\mathcal{H}$'s perspective, who, like a person under global anesthesia, has no sense of time during its sleep and hence can think that the wake-up events that it calls the beginning of a clock cycle happen at a constant rate, $\mathcal{E}$ acts as an adversary who can make any finite number of moves during a clock cycle (i.e. while $\mathcal{H}$ was sleeping), just as the HPM-model assumes. This scenario uniquely determines an e-computation branch $B_{\mathcal{E}}$ of $\mathcal{E}$ that we call the $(\mathcal{E}, e, \mathcal{H})$-branch, and an e-computation branch $B_{\mathcal{H}}$ of $\mathcal{H}$ that we call the $(\mathcal{H}, e, \mathcal{E})$-branch. What we call the $\mathcal{H}$ vs. $\mathcal{E}$ run on e is the run generated in this play. In particular—since we let $\mathcal{H}$ play in the role of $\top$—this is the run spelled by $B_{\mathcal{H}}$. $\mathcal{E}$, who plays in the role of $\bot$, sees the same run, only it sees the labels of that run in negative colors. That is, $B_{\mathcal{E}}$ cospells rather than spells that run. This is exactly what clause (a) of Lemma 3.3 asserts. Now suppose $e_0, e_1, e_2, \ldots$ and g are as in clause (b), and e is one of the e_c. Then, using g, the contents of any initial segment of the valuation tape(s) can be effectively constructed from c. Therefore the work of either machine can be effectively traced up to any given computation step n, which implies clause (b).

4. Operations on computational problems

As noted, computability logic is an approach that uses logical formalism for specifying and studying interactive computational problems in a systematic way, understanding logical operators as operations on games/problems. It is time to define basic operations on games. It should be noted that even though our interests are focused on static games, the operations defined in this section are equally meaningful for non-static (*dynamic*) games as well. So, we do not restrict the scope of those definitions to static games, and throughout the section we let the letters A, B range over any games. Here comes the first definition:

Definition 4.1. Let A be any game, $x_1, \ldots, x_n$ $(n \geqslant 0)$ pairwise distinct variables, and $t_1, \ldots, t_n$ any terms. For any valuation e, let e° denote the unique valuation that agrees with e on all variables that are not among $x_1, \ldots, x_n$, such that, for each $x_i \in \{x_1, \ldots, x_n\}$, $e^\circ(x_i) = e(t_i)$. Then we define the game $A[x_1/t_1, \ldots, x_n/t_n]$ by stipulating that, for any valuation e, $\mathbf{Lr}_e^{A[x_1/t_1,\ldots,x_n/t_n]} = \mathbf{Lr}_{e^\circ}^A$ and $\mathbf{Wn}_e^{A[x_1/t_1,\ldots,x_n/t_n]} = \mathbf{Wn}_{e^\circ}^A$; in other words, $e[A[x_1/t_1, \ldots, x_n/t_n]] = e^\circ[A]$.

This operation, that we call *substitution of variables*, is a generalization of the standard operation of substitution of variables known from classical predicate logic. Intuitively, $A[x_1/t_1, \ldots, x_n/t_n]$ is the same as A, only with (the values of) variables $x_1, \ldots, x_n$ "read as'' (the values of) $t_1, \ldots, t_n$, respectively. Each t_i here can be either a variable or a constant. Remember from Section 2 that when t_i is a constant, $e(t_i) = t_i$.

Example: If A is the elementary game $x \times y > z + u$, then $A[x/z, z/6, u/y]$ would be the game $z \times y > 6 + y$.

Sometimes it is convenient to fix a certain tuple $(x_1, \ldots, x_n)$ of pairwise distinct variables for a game A throughout a context and write A in the form $A(x_1, \ldots, x_n)$. We will refer to such a tuple $(x_1, \ldots, x_n)$ as the *attached tuple* of (the given representation of) A. When doing so, we do not necessarily mean that $A(x_1, \ldots, x_n)$ is an n-ary game and/or that

[3] Clause (b) of our Lemma 3.3 is slightly stronger than the official formulation of the corresponding clause (c) of Lemma 20.4 of [10]. However, the proof given in [10] is just as good for our present strong formulation.

$x_1, \ldots, x_n$ are exactly the variables on which this game depends. Once A is given with an attached tuple $(x_1, \ldots, x_n)$, we will write $A(t_1, \ldots, t_n)$ to mean the same as the more clumsy expression $A[x_1/t_1, \ldots, x_n/t_n]$. A similar notational practice is common in the literature for predicates. Thus, the above game $x \times y > z + u$ can be written as $A(x, z, u)$, in which case $A(z, 6, y)$ will denote the game $z \times y > 6 + y$.

We have already seen two meanings of symbol $\neg$: one was that of an operation on players (Section 2), and one that of an operation on runs (Section 3). Here comes the third meaning of it—that of an operation on games:

Definition 4.2. The *negation* $\neg A$ of a game A is defined by:
- $\mathbf{Lr}_e^{\neg A} = \{\Gamma \mid \neg\Gamma \in \mathbf{Lr}_e^A\}$.
- $\mathbf{Wn}_e^{\neg A}\langle\Gamma\rangle = \neg\mathbf{Wn}_e^A\langle\neg\Gamma\rangle$.

Intuitively, $\neg A$ is game A with the roles of the two players switched: $\top$'s moves or wins become $\bot$'s moves or wins, and vice versa. For example, if *Chess* is the game of chess from the point of view of the white player, then $\neg$*Chess* would be the same game from the point of view of the black player.

The operations $\wedge$ and $\vee$ produce parallel combinations of games. Playing $A_1 \wedge \cdots \wedge A_n$ or $A_1 \vee \cdots \vee A_n$ means playing the n games concurrently. Both $A_1 \wedge \cdots \wedge A_n$ and $A_1 \vee \cdots \vee A_n$ have exactly the same structure (legal moves), and the only difference is in how the winner is determined: in order to win, in the former $\top$ needs to win in each of the n components, while in the latter winning in one of the components is sufficient. To indicate that a given move is made in the ith component, the player should prefix it with the string "$i.$''. Any move that does not have one of such prefixes will be considered illegal.

Here comes the formal definition. In it the notation Γ^γ means the result of removing from Γ all labeled moves except those of the form $\wp\gamma\alpha$ ($\wp \in \{\top, \bot\}$), and then deleting the prefix "γ'' in the remaining moves, i.e. changing each such $\wp\gamma\alpha$ to $\wp\alpha$.

Definition 4.3. Let $A_1, \ldots, A_n$ ($n \geqslant 2$) be any games.

The *parallel conjunction* $A_1 \wedge \cdots \wedge A_n$ of $A_1, \ldots, A_n$ is defined by:
- $\Gamma \in \mathbf{Lr}_e^{A_1\wedge\cdots\wedge A_n}$ iff every move of Γ has one of the prefixes "1.'',..., "$n.$'' and, for each $i \in \{1, \ldots, n\}$, $\Gamma^{i\cdot} \in \mathbf{Lr}_e^{A_i}$.
- Whenever $\Gamma \in \mathbf{Lr}_e^{A_1\wedge\cdots\wedge A_n}$, $\mathbf{Wn}_e^{A_1\wedge\cdots\wedge A_n}\langle\Gamma\rangle = \top$ iff, for all $i \in \{1, \ldots, n\}$, $\mathbf{Wn}_e^{A_i}\langle\Gamma^{i\cdot}\rangle = \top$.

The *parallel disjunction* $A_1 \vee \cdots \vee A_n$ of $A_1, \ldots, A_n$ is defined in exactly the same way, only with "$\top$'' replaced by "$\bot$''. Equivalently, it can be defined by $A_1 \vee \cdots \vee A_n \ =_{def}\ \neg(\neg A_1 \wedge \cdots \wedge \neg A_n)$.

The other operation from the same group—the *parallel implication* $A \to B$ of games A and B—is defined by $A \to B \ =_{def}\ (\neg A) \vee B$.

Intuitively, $A \to B$ is the problem of *reducing* B (*consequent*) to A (*antecedent*). That is, solving $A \to B$ means solving B having A as a *computational resource*. Generally, computational resources are symmetric to computational tasks/problems: what is a problem for one player to solve, is a resource for the other player to use, and vice versa. Since in the antecedent of $A \to B$ the roles of the players are switched, A becomes a problem for $\bot$ to solve, and hence a resource that $\top$ can use. Thus, our semantics of computational problems is, at the same time, a semantics of computational resources. As noted before, this offers a materialization of the abstract resource philosophy associated with linear logic [4]. We will see a couple of examples later supporting this claim. More elaborated examples and discussions can be found in [10], where, in Section 26, the context of computational resources is further extended to informational and physical resources as well. Ref. [12] also abounds with illustrative examples.

On an intuitive level, our parallel operations can be related to the corresponding multiplicative operators of linear logic. The game-semantical approach to linear-logic-style connectives is not new in principle, even if it has rather stubbornly resisted a complete treatment within natural frameworks. What makes our understanding of "multiplicatives'' substantially different from other ([1,3,6] etc.) interpretations is that they are *free*, i.e. generate free games, even when applied to strict games. As an example, consider the game *Chess* $\wedge$ *Chess*. Assume an agent plays this two-board game over the Internet against two independent adversaries—adversary #1 on board #1 and adversary #2 on board #2—that, together, form the (one) environment for the agent. As we agreed, *Chess* is the game playing which means playing the game of chess white. Hence, in the initial position of *Chess* $\wedge$ *Chess*, only the agent has legal moves. But once such a move is made, say, on board #1, the picture changes. Now both the agent and the environment have legal moves: the agent may make another opening move on board #2, while the environment—in particular adversary #1—may make

a reply move on board #1. This is a situation where which player 'is to move' is no longer strictly determined, so the next player to move will be the one who can or wants to act faster. A strict-game approach would impose some additional conditions uniquely determining the next player to move. Such conditions would most likely be artificial and not quite adequate, for the situation we are trying to model is a concurrent play on two boards against two independent adversaries, and we cannot or should not expect any coordination between their actions. Most of the compound tasks we perform in everyday life are free rather than strict, and so are most computer communication/interaction protocols. A strict understanding of $\wedge$ would essentially mean some sort of an (in a way interspersed but still) sequential rather than truly parallel/concurrent combination of tasks, where no steps in one component would be allowed to be made until receiving a response in the other component, contrary to the very spirit of the idea of parallel/distributed computation.

It is no accident that we use classical symbols for the above operations. As this is easy to see, the meanings of these operations, as well as the meanings of the so called blind quantifiers $\forall$, $\exists$ that will be defined shortly, are exactly classical when their scope is restricted to elementary games (in which case the compound games generated by these operations also remain elementary). This is what makes classical logic just a special fragment of the more general and expressive computability logic. Once the scope of the "classical'' propositional connectives is extended beyond elementary games, however, their behavior starts resembling that of the multiplicative operators of linear logic. For example, the principle $A \rightarrow A \wedge A$ generally fails for non-elementary games. Yet, this resemblance is rather shallow, and typically disappears as soon as we start considering longer and "deeper'' formulas. See [10] for more about how computability logic relates to linear logic. We do not want to go into details of this discussion here because, as pointed out in Section 1, this work is not at all an attempt to find a justification for linear logic—the popular logic that is syntactically so appealing yet lacks a convincing semantics.

The next group of operations: $\sqcap$, $\sqcup$, $\bigsqcap$ and $\bigsqcup$ that we call *choice* operations, on the other hand, bear resemblance with the additive operators of linear logic. Based on their semantics, in more traditional terms they can be characterized as *constructive* versions of conjunction, disjunction, universal quantifier and existential quantifier, respectively.

$\bigsqcap x A(x)$ is the game where, in the initial position, only $\bot$ has legal moves. Such a move consists in a choice of one of the elements of the universe of discourse. After $\bot$ makes a move $c \in \{0, 1, \ldots\}$, the game continues (and the winner is determined) according to the rules of $A(c)$. If no initial move is made, $\top$ is considered the winner as there was no particular (sub)problem specified by $\bot$ that $\top$ failed to solve. $A \sqcap B$ is similar, only here the choice is just made between "left'' ("1'') and "right'' ("2''). $\bigsqcup$ and $\sqcup$ are symmetric to $\bigsqcap$ and $\sqcap$, with the only difference that now it is $\top$ rather than $\bot$ who makes an initial move/choice. Here is the formal definition:

Definition 4.4. Let $A(x)$, $A_1, \ldots, A_n$ $(n \geqslant 2)$ be any games.

The *choice conjunction* $A_1 \sqcap \cdots \sqcap A_n$ of $A_1, \ldots, A_n$ is defined by:

- $\mathbf{Lr}_e^{A_1 \sqcap \cdots \sqcap A_n} = \{\langle\rangle\} \cup \{\langle \bot i, \Delta\rangle \mid i \in \{1, \ldots, n\},\ \Delta \in \mathbf{Lr}_e^{A_i}\}$.
- $\mathbf{Wn}_e^{A_1 \sqcap \cdots \sqcap A_n}\langle\Gamma\rangle = \bot$ iff $\Gamma = \langle \bot i, \Delta\rangle$, where $i \in \{1, \ldots, n\}$ and $\mathbf{Wn}_e^{A_i}\langle\Delta\rangle = \bot$.

The *choice disjunction* $A_1 \sqcup \cdots \sqcup A_n$ of $A_1, \ldots, A_n$ is defined in exactly the same way, only with "$\top$'' instead of "$\bot$''. Equivalently, it can be defined by $A_1 \sqcup \cdots \sqcup A_n \ =_{def}\ \neg(\neg A_1 \sqcap \cdots \sqcap \neg A_n)$.

The *choice universal quantification* $\bigsqcap x A(x)$ of $A(x)$ is defined by:

- $\mathbf{Lr}_e^{\bigsqcap x A(x)} = \{\langle\rangle\} \cup \{\langle \bot c, \Delta\rangle \mid c \text{ is a constant}, \Delta \in \mathbf{Lr}_e^{A(c)}\}$.
- $\mathbf{Wn}_e^{\bigsqcap x A(x)}\langle\Gamma\rangle = \bot$ iff $\Gamma = \langle \bot c, \Delta\rangle$, where c is a constant and $\mathbf{Wn}_e^{A(c)}\langle\Delta\rangle = \bot$.

The *choice existential quantification* $\bigsqcup x A(x)$ of $A(x)$ is defined in exactly the same way, only with "$\top$'' instead of "$\bot$''. Equivalently, it can be defined by $\bigsqcup x A(x) \ =_{def}\ \neg\bigsqcap x \neg A(x)$.

A few examples would help. The problem of computing a function f can be specified as $\bigsqcap x \bigsqcup y(f(x) = y)$. This is a game of depth 2, where the first legal move—selecting a particular value k for x—is by $\bot$. Making such a move brings the game down to $\bigsqcup y(f(k) = y)$. The second move—selecting a value n for y—is by $\top$, after which the game continues (or rather stops) as $f(k) = n$. The latter is an elementary game won by $\top$ iff $f(k)$ really equals n. Obviously f is computable in the standard sense iff $\bigsqcap x \bigsqcup y(f(x) = y)$ is winnable, i.e. computable in our sense.

Next, the problem of deciding a predicate $p(x)$ would be specified as $\bigsqcap x(p(x) \sqcup \neg p(x))$. This is the game where, after $\bot$ selects a value k for x, the machine should reply by one of the moves 1 or 2; the game will be considered won by the machine if $p(k)$ is true and the move 1 was made, or $p(k)$ is false and the choice was 2, so that decidability of $p(x)$ means nothing but existence of a machine that wins the game $\bigsqcap x(p(x) \sqcup \neg p(x))$.

To get a feel of $\rightarrow$ as a problem reduction operation, let us consider reduction of the acceptance problem to the halting problem (the example borrowed from [10]). The halting problem can be expressed by $\sqcap x \sqcap y(Halts(x, y) \sqcup \neg Halts(x, y))$, where $Halts(x, y)$ is the predicate "Turing machine x halts on input y''. Similarly, the acceptance problem can be expressed by the formula $\sqcap x \sqcap y(Accepts(x, y) \sqcup \neg Accepts(x, y))$, where $Accepts(x, y)$ is the predicate "Turing machine x accepts input y''. While the acceptance problem is not decidable, it is algorithmically reducible to the halting problem. In particular, there is an HPM that wins

$$\sqcap x \sqcap y(Halts(x, y) \sqcup \neg Halts(x, y)) \rightarrow \sqcap x \sqcap y(Accepts(x, y) \sqcup \neg Accepts(x, y)).$$

Here is a strategy for solving this problem: Wait till the environment selects values k and n for x and y in the consequent (if such a selection is never made, the machine wins). Then specify the same values k and n for x and y in the antecedent (where the roles of the machine and the environment are switched), and see whether $\bot$ responds by 1 or 2 there. If the response is 1, simulate machine k on input n until it halts, and select, in the consequent, 1 or 2 depending on whether the simulation accepted or rejected. And if $\bot$'s response in the antecedent was 2, then select 2 in the consequent.

We can see that what the machine did in the above strategy indeed was a reduction: it used an (external) solution to the halting problem to solve the acceptance problem. There are various natural concepts of reduction, and the sort of reduction captured by $\rightarrow$, that we call *linear reduction*, is most basic among them.

One of the other, well-established, concepts of reduction is *mapping reduction*: a predicate $p(x)$ is said to be mapping reducible to a predicate $q(x)$ iff there is an effective function f such that, for any constant c, $p(c)$ is true iff $q(f(c))$ is true. Using $A \leftrightarrow B$ as an abbreviation for $(A \rightarrow B) \wedge (B \rightarrow A)$, it is not hard to see that mapping reducibility of $p(x)$ to $q(x)$ means nothing but winnability of the game $\sqcap x \sqcup y(p(x) \leftrightarrow q(y))$.

Notice that while standard approaches only allow us to talk about (a whatever sort of) reducibility as a *relation* between problems, in our approach reduction becomes an *operation* on problems, with reducibility as a relation simply meaning computability of the corresponding combination (such as $\sqcap x \sqcup y(p(x) \leftrightarrow q(y))$ or $A \rightarrow B$) of problems. Similarly for other relations or properties such as the property of *decidability*. The latter becomes the operation of *deciding* if we define the problem of deciding a predicate (or any computational problem) $p(x)$ as the game $\sqcap x(p(x) \sqcup \neg p(x))$. So, now we can meaningfully ask questions such as "Is the linear reduction of the problem of deciding $p(x)$ to the problem of deciding $q(x)$ linearly reducible to the mapping reduction of $p(x)$ to $q(x)$?''. This question would be equivalent to whether the following problem is (always) computable:

$$\sqcap x \sqcup y(p(x) \leftrightarrow q(y)) \rightarrow (\sqcap x(q(x) \sqcup \neg q(x)) \rightarrow \sqcap x(p(x) \sqcup \neg p(x))). \tag{1}$$

This problem is indeed computable no matter what particular predicates $p(x)$ and $q(x)$ are, which means that mapping reduction is at least as strong as linear reduction. Here is a strategy: wait till $\bot$ selects a value k for x in the consequent of the consequent of (1). Then specify the same value k for x in the antecedent of (1), and wait till $\bot$ replies there by selecting a value n for y. Then select the same value n for x in the antecedent of the consequent of (1). $\bot$ will have to respond by 1 or 2 in that component of the game. Repeat that very response in the consequent of the consequent of (1), and celebrate victory.

Expression (1) is a legal formula of the language of **CL3** which, according to our main Theorem 5.9, is sound and complete with respect to computability semantics. So, had our ad hoc methods failed to find an answer (and this would certainly be the case if we dealt with a more complex problem), the existence of a successful algorithmic strategy could have been established by showing that (1) is provable in **CL3**. Moreover, by clause (a) of Theorem 5.9, after finding an **CL3**-proof of (1), we would not only know that an algorithmic solution to (1) exists, but we would also be able to constructively obtain such a solution from the proof. On the other hand, the fact that linear reduction is not as strong as mapping reduction could be established by showing that **CL3** does not prove

$$(\sqcap x(q(x) \sqcup \neg q(x)) \rightarrow \sqcap x(p(x) \sqcup \neg p(x))) \rightarrow \sqcap x \sqcup y(p(x) \leftrightarrow q(y)). \tag{2}$$

This negative fact, too, can be established effectively as, according to Theorem 5.7, the relevant fragment of **CL3** is decidable. Our proof of the completeness part of Theorem 5.9 would then offer a way how to construct particular predicates $p(x)$ and $q(x)$ for which (2) is not computable.

These few examples must be sufficient to provide insights into the utility of computability logic and **CL3** in particular for theory of computing: our logic offers a convenient tool for asking and answering questions in the above style (and beyond) in a systematic way, something that so far has been mostly done in an ad hoc manner or has been simply

impossible to do. By iterating available operators, we can express and explore an infinite variety computational problems and relations between them, only few of which may have special names established in the literature.

The next, already mentioned group of operations is what we call "blind quantifiers'': $\forall$ and $\exists$, with hardly any reasonably close counterparts in linear logic. For certain reasons, the operations $\forall x$ and $\exists x$ we only define for games called x-unistructural. A game A is said to be *x-unistructural* (or *unistructural in x*) iff, for any two valuations e_1 and e_2 that agree on all variables except perhaps x, we have $\mathbf{Lr}^A_{e_1} = \mathbf{Lr}^A_{e_2}$. Intuitively, this is a game whose structure does not depend on x. In fact whether we impose the x-unistructurality condition or not is irrelevant in our present case because this condition is automatically satisfied anyway: as shown in [10], all games that can be expressed in the language of **CL3** are unistructural, and obviously all unistructural games are also x-unistructural.

Definition 4.5. Let x be any variable and $A(x)$ any x-unistructural game.

The *blind universal quantification* $\forall x A(x)$ *of* $A(x)$ is defined by:

- $\mathbf{Lr}_e^{\forall x A(x)} = \mathbf{Lr}_e^{A(x)}$.
- $\mathbf{Wn}_e^{\forall x A(x)}\langle\Gamma\rangle = \top$ iff, for every constant c, $\mathbf{Wn}_e^{A(c)}\langle\Gamma\rangle = \top$.

The *blind existential quantification* $\exists x A(x)$ *of* $A(x)$ is defined in exactly the same way, only with "$\bot$'' instead of "$\top$''. Equivalently, it can be defined by $\exists x A(x) =_{def} \neg\forall x\neg A(x)$.

The meaning of $\forall x A(x)$ is similar to that of $\bigsqcap x A(x)$, with the difference that $\bot$ does not make a move specifying a particular value of x, so that $\top$ has to play blindly in a way that would be successful for any possible value of x. Alternatively, $\forall x A(x)$ can be thought of as the version of $\bigsqcap x A(x)$ where the particular value of x selected by $\bot$ remains invisible to $\top$. This way, $\forall$ and $\exists$ produce games with *imperfect information*. Compare the problems $\bigsqcap x(Even(x) \sqcup Odd(x))$ and $\forall x(Even(x) \sqcup Odd(x))$. The former is an easy-to-compute problem of depth 2, while the latter is an incomputable problem of depth 1 with only by the machine to make a move—select the true disjunct, which is hardly possible to do as the value of x remains unspecified. Some problems that depend on x can be however solved having only partial information on x. For example, in order to tell whether x is even or odd, we do not really need to read the whole (decimal representation of) x—it would be sufficient to look at its last digit, i.e. know the value of (x **Mod** 10). Thus, the problem $\forall x(\bigsqcup y(y = (x \textbf{ Mod } 10)) \rightarrow (Even(x) \sqcup Odd(x)))$ is computable, which is a more informative statement than if we had stated computability of the same problem with $\bigsqcap$ instead of $\forall$. As noted a while ago, the meanings of $\forall$ and $\exists$ are exactly classical when applied to elementary games, which explains why we use the classical notation for them.

Another important group of operations comprises *recurrence operations*. They come in different flavors (see [12]), perhaps the most interesting of which is what is called *branching recurrence*[4] $\mathbin{\downarrow\!\circ}$. Intuitively $\mathbin{\downarrow\!\circ} A$, as a resource, is A that can be reused an arbitrary number of times. The same is true for the other sorts of recurrences, but there are several natural understandings of reusage, with $\mathbin{\downarrow\!\circ}$ corresponding to the strongest form of it and the other recurrence operations corresponding to weaker concepts of reusage/recycling (and it is not clear which of the recurrence operations best corresponds to what the exponential operator ! of linear logic was meant to capture). The operation $\circ\!\!-$, called *weak reduction*,[5] is defined by $A \circ\!\!- B = \mathbin{\downarrow\!\circ} A \rightarrow B$. This operation formalizes our weakest possible intuitive concept of reduction. The difference between $\rightarrow$ and $\circ\!\!-$ as reduction operations is that while in the former every act of resource (antecedent) consumption is strictly accounted for, the latter allows uncontrolled usage of resources. One of the conjectures stated in [10] is that we get exactly intuitionistic calculus when the intuitionistic implication is understood as $\circ\!\!-$ and the other intuitionistic operators understood as the corresponding choice operations. Recurrence operators and weak reduction are not in the logical vocabulary of **CL3**, and hence we do not give formal definitions for them. Such definitions can be found in [10,12].

According to Theorem 14.1 of [10], all of the operations that we discussed in this section preserve the static and unistructural properties of games. Taking into account that predicates as elementary games are static (Proposition 2.3) and obviously unistructural, their closure under those operations forms a natural class of unistructural computational problems. All of those operations except $\mathbin{\downarrow\!\circ}$ and $\circ\!\!-$ also preserve the finite-depth property. Hence the closure of the set of all elementary problems under substitution of variables, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$, $\sqcap$, $\sqcup$, $\bigsqcap$ and $\bigsqcup$ forms a natural class of finite-depth, unistructural computational problems. As we are going to see, this is exactly the class of problems

[4] In [10] this operation is called *branching conjunction* and is denoted by !.

[5] Ref. [10] uses the symbol $\Rightarrow$ for this operation.

expressible in the language of **CL3**. Finally, as already noted more than once, the operations $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$ preserve the elementary property of games: they send predicates to predicates; and, when restricted to predicates, they coincide with the same-name classical operations. Of course, the same applies to the operation of substitution of variables, as well as the trivial games $\bot$ and $\top$ that can be understood as 0-ary operations on games.

One more game operation that we are going to look at is that of prefixation, which is somewhat reminiscent of the modal operator(s) of dynamic logic. This operation takes two arguments: a game A and a position Φ that must be what we call a unilegal position of A (otherwise the operation is undefined). Γ is said to be a *unilegal run* (position if finite) of a game A iff, for every valuation e, $\Gamma \in \mathbf{Lr}_e^A$. As noted above, all games that we deal with in this paper are unistructural, and for such games obviously there is no difference between "unilegal'' and "legal''.

Definition 4.6. Assume Φ is a unilegal position of a game A. The *Φ-prefixation* of A, denoted $\langle\Phi\rangle A$, is defined as follows:

- $\mathbf{Lr}_e^{\langle\Phi\rangle A} = \{\Gamma \mid \langle\Phi, \Gamma\rangle \in \mathbf{Lr}_e^A\}$.
- $\mathbf{Wn}_e^{\langle\Phi\rangle A}\langle\Gamma\rangle = \mathbf{Wn}_e^A\langle\Phi, \Gamma\rangle$.

Intuitively, $\langle\Phi\rangle A$ is the game playing which means playing A starting (continuing) from position Φ. That is, $\langle\Phi\rangle A$ is the game to which A evolves (is "brought down'') after the (lab)moves of Φ have been made. We have already used this intuition when explaining the meaning of the choice operations. For example, we said that after $\bot$ makes an initial move c, the game $\bigsqcap x A(x)$ continues as $A(c)$. What this meant was nothing but that $\langle\bot c\rangle(\bigsqcap x A(x)) = A(c)$. The following proposition summarizes this sort of a characterization of the choice operations, and extends it to the other operations, too. It tells us what the legal initial moves for a given game are, and to what game that game evolves after such a (uni)legal move is made.

Proposition 4.7. *In each of the following clauses, e is any valuation, $\wp$ either player, and α, β any moves; in each subclause* (b)*, the game on the left of the equation is assumed to be defined, $i \in \{1, \ldots, n\}$ and $c \in \{0, 1, 2, \ldots\}$:*

(1) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{\neg A}$ *iff* $\langle\neg\wp\alpha\rangle \in \mathbf{Lr}_e^A$,
(b) $\langle\wp\alpha\rangle\neg A = \neg(\langle\neg\wp\alpha\rangle A)$.

(2) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{A_1\wedge\cdots\wedge A_n}$ *iff* $\alpha = i.\beta$, *where* $i \in \{1, \ldots, n\}$ *and* $\langle\wp\beta\rangle \in \mathbf{Lr}_e^{A_i}$,
(b) $\langle\wp i.\beta\rangle(A_1 \wedge \cdots \wedge A_n) = A_1 \wedge \cdots \wedge A_{i-1} \wedge \langle\wp\beta\rangle A_i \wedge A_{i+1} \wedge \cdots \wedge A_n$.

(3) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{A_1\vee\cdots\vee A_n}$ *iff* $\alpha = i.\beta$, *where* $i \in \{1, \ldots, n\}$ *and* $\langle\wp\beta\rangle \in \mathbf{Lr}_e^{A_i}$,
(b) $\langle\wp i.\beta\rangle(A_1 \vee \cdots \vee A_n) = A_1 \vee \cdots \vee A_{i-1} \vee \langle\wp\beta\rangle A_i \vee A_{i+1} \vee \cdots \vee A_n$.

(4) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{A\rightarrow B}$ *iff* $\alpha = i.\beta$, *where* $\begin{cases} i = 1 \text{ and } \langle\neg\wp\beta\rangle \in \mathbf{Lr}_e^A, \text{ or} \\ i = 2 \text{ and } \langle\wp\beta\rangle \in \mathbf{Lr}_e^B, \end{cases}$
(b) $\begin{cases} \langle\wp 1.\beta\rangle(A \rightarrow B) = \langle\neg\wp\beta\rangle A \rightarrow B, \\ \langle\wp 2.\beta\rangle(A \rightarrow B) = A \rightarrow \langle\wp\beta\rangle B. \end{cases}$

(5) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{A_1\sqcap\cdots\sqcap A_n}$ *iff* $\wp = \bot$ *and* $\alpha = i \in \{1, \ldots, n\}$,
(b) $\langle\bot i\rangle(A_1 \sqcap \cdots \sqcap A_n) = A_i$.

(6) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{A_1\sqcup\cdots\sqcup A_n}$ *iff* $\wp = \top$ *and* $\alpha = i \in \{1, \ldots, n\}$,
(b) $\langle\top i\rangle(A_1 \sqcup \cdots \sqcup A_n) = A_i$.

(7) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{\bigsqcap x A(x)}$ *iff* $\wp = \bot$ *and* $\alpha = c \in \{0, 1, 2, \ldots\}$,
(b) $\langle\bot c\rangle\bigsqcap x A(x) = A(c)$.

(8) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{\bigsqcup x A(x)}$ *iff* $\wp = \top$ *and* $\alpha = c \in \{0, 1, 2, \ldots\}$,
(b) $\langle\top c\rangle\bigsqcup x A(x) = A(c)$.

(9) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{\forall x A(x)}$ *iff* $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{A(x)}$,
(b) $\langle\wp\alpha\rangle\, \forall x A(x) = \forall x \langle\wp\alpha\rangle A(x)$.

(10) (a) $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{\exists x A(x)}$ *iff* $\langle\wp\alpha\rangle \in \mathbf{Lr}_e^{A(x)}$,
(b) $\langle\wp\alpha\rangle\, \exists x A(x) = \exists x \langle\wp\alpha\rangle A(x)$.

The above fact is known from [10]. Its proof consists in just a routine analysis of the relevant definitions.

Just like this is the case with recurrence operations, the language of **CL3** does not have any constructs corresponding to prefixation. However, this operation will be heavily exploited in our soundness and completeness proof for **CL3** in Part 2. Generally, prefixation is very handy in visualizing a (unilegal) run of a given game A. In particular, every (sub)position Φ of such a run can be represented by, or thought of as, the game $\langle\Phi\rangle A$.

Here is an example. Remember game (1). Based on Proposition 4.7, the run $\langle\bot 2.2.7, \top 1.7, \bot 1.9, \top 2.1.9, \bot 2.1.1, \top 2.2.1\rangle$ is a (uni)legal run of that game, and to it corresponds the following sequence of games:

(i) $(\bigsqcap x\bigsqcup y(p(x) \leftrightarrow q(y))) \rightarrow (\bigsqcap x(q(x) \sqcup \neg q(x)) \rightarrow \bigsqcap x(p(x) \sqcup \neg p(x)))$,
i.e. $\langle\rangle(1)$;

(ii) $(\bigsqcap x\bigsqcup y(p(x) \leftrightarrow q(y))) \rightarrow (\bigsqcap x(q(x) \sqcup \neg q(x)) \rightarrow (P(7) \sqcup \neg p(7)))$,
i.e. $\langle\bot 2.2.7\rangle$(i), i.e. $\langle\bot 2.2.7\rangle(1)$;

(iii) $(\bigsqcup y(p(7) \leftrightarrow q(y))) \rightarrow (\bigsqcap x(q(x) \sqcup \neg q(x)) \rightarrow (p(7) \sqcup \neg p(7)))$,
i.e. $\langle\top 1.7\rangle$(ii), i.e. $\langle\bot 2.2.7, \top 1.7\rangle(1)$;

(iv) $(p(7) \leftrightarrow q(9)) \rightarrow (\bigsqcap x(q(x) \sqcup \neg q(x)) \rightarrow (p(7) \sqcup \neg p(7)))$,
i.e. $\langle\bot 1.9\rangle$(iii), i.e. $\langle\bot 2.2.7, \top 1.7, \bot 1.9\rangle(1)$;

(v) $(p(7) \leftrightarrow q(9)) \rightarrow ((q(9) \sqcup \neg q(9)) \rightarrow (p(7) \sqcup \neg p(7)))$,
i.e. $\langle\top 2.1.9\rangle$(iv), i.e. $\langle\bot 2.2.7, \top 1.7, \bot 1.9, \top 2.1.9\rangle(1)$;

(vi) $(p(7) \leftrightarrow q(9)) \rightarrow (q(9) \rightarrow (p(7) \sqcup \neg p(7)))$,
i.e. $\langle\bot 2.1.1\rangle$(v), i.e. $\langle\bot 2.2.7, \top 1.7, \bot 1.9, \top 2.1.9, \bot 2.1.1\rangle(1)$;

(vii) $(p(7) \leftrightarrow q(9)) \rightarrow (q(9) \rightarrow p(7))$,
i.e. $\langle\top 2.2.1\rangle$(vi), i.e. $\langle\bot 2.2.7, \top 1.7, \bot 1.9, \top 2.1.9, \bot 2.1.1, \top 2.2.1\rangle(1)$.

Player $\top$ is the winner because the run hits a true elementary game. In this run $\top$ has followed the winning strategy that we described for (1) earlier.

We finish this section by reproducing a fact proven in [10, Proposition 21.3], according to which modus ponens preserves computability, and does so in a constructive sense:

Proposition 4.8. *For any computational problems A and B, if $\vDash A$ and $\vDash A \rightarrow B$, then $\vDash B$. Moreover, there is an effective procedure that converts any two HPMs $\mathcal{H}_1$ and $\mathcal{H}_2$ into an HPM $\mathcal{H}_3$ such that, for any computational problems A, B and any valuation e, whenever $\mathcal{H}_1 \vDash_e A$ and $\mathcal{H}_2 \vDash_e A \rightarrow B$, we have $\mathcal{H}_3 \vDash_e B$.*

A similar closure property was proven in Section 21 of [10] with respect to the rules $A \mapsto \bigsqcap x A$ and $A \mapsto \mathbin{!} A$, with $\mathcal{P} \mapsto C$ here and later meaning "from premise(s) $\mathcal{P}$ conclude C"'.

5. Logic CL3

By the *classical language* we mean the language of pure classical first-order logic with individual constants but without equality and functional symbols. We assume that the set of *terms*—i.e. *variables* and *constants*—of this language is the same as the one we fixed in Section 2. As always, each *predicate letter* comes with a fixed *arity*. An (n-ary non-logical) *atom* is the expression $p(t_1, \ldots, t_n)$, where p is an n-ary predicate letter and the t_i are terms.

The language of **CL3** extends the classical language by adding the operators $\sqcap$, $\sqcup$, $\bigsqcap$, $\bigsqcup$ to its vocabulary. The names that we use for the logical operators of the language are the same as for the (same-symbolic-name) game operations defined in the previous section. Throughout the rest of this paper by a *formula* we mean a formula of this language. The definition is standard—the set of formulas is the smallest set of expressions such that:

- Non-logical atoms and the *logical atoms* $\top$ and $\bot$ are formulas;
- If $F_1, \ldots, F_n$ ($n \geqslant 2$) are formulas, then so are $\neg F_1$, $F_1 \wedge \cdots \wedge F_n$, $F_1 \vee \cdots \vee F_n$, $F_1 \rightarrow F_2$, $F_1 \sqcap \cdots \sqcap F_n$, $F_1 \sqcup \cdots \sqcup F_n$;
- If F is a formula and x is a variable, then $\forall x F$, $\exists x F$, $\bigsqcap x F$, $\bigsqcup x F$ are formulas.

The definitions of what a *free* or *bound* occurrence of a variable means are also standard, keeping in mind that now a variable can be bound by any of the four *quantifiers* $\forall$, $\exists$, $\bigsqcap$, $\bigsqcup$. Every occurrence of a constant also counts as *free*. By a *free variable* of a formula F we mean a variable that has free occurrences in F. Similarly, the *free terms* of F are its free variables plus the constants occurring in F. As known, *classical validity* of a formula of the classical language that contains constants means the same as validity of the same formula with its constants understood as free variables.

So, for a reader more used to the version of classical logic where variables are the only terms, it is perfectly safe to think of constants as free variables.

Furthermore, for known reasons that equally apply to both classical and computability logics, there is no loss of expressive power if the scope of the term *formula* is narrowed down to expressions in which no quantifier binds a variable that also has free occurrences within the same expression. Indeed, if this condition is violated, nothing would be easier than to just rename variables. So, we agree that, from now on, the term "formula" will be exclusively understood in this restricted sense, unless otherwise suggested by the context.

In the previous section, substitution of variables was defined as an operation on games. Here we define a "similar'' operation on formulas called *substitution of terms*. Suppose F is a formula, $t_1, \ldots, t_n$ are pairwise distinct terms, and $t'_1, \ldots, t'_n$ are any terms not bound in F. Then $F[t_1/t'_1, \ldots, t_n/t'_n]$ stands for the result of simultaneously substituting in F all free occurrences of $t_1, \ldots, t_n$ by $t'_1, \ldots, t'_n$, respectively.

In concordance with a similar notational practice established in Section 4 for games, sometimes we represent a formula F as $F(t_1, \ldots, t_n)$ where the t_i are pairwise distinct terms. In the context defined by such a representation, $F(t'_1, \ldots, t'_n)$ will mean the same as $F[t_1/t'_1, \ldots, t_n/t'_n]$. Our disambiguating convention is that the context is set by the expression that was used earlier. That is, when we first mention $F(t_1, \ldots, t_n)$ and only after that use the expression $F(t'_1, \ldots, t'_n)$, the latter should be understood as $F[t_1/t'_1, \ldots, t_n/t'_n]$ rather than the former understood as $F[t'_1/t_1, \ldots, t'_n/t_n]$. It should be noted that, when representing F as $F(t_1, \ldots, t_n)$, we do not necessarily mean that $t_1, \ldots, t_n$ are exactly the free terms of F.

An *interpretation* is a function * that sends each n-ary predicate letter p to an elementary game $p^* = A(x_1, \ldots, x_n)$ with an attached n-tuple of (pairwise distinct) variables. This assignment extends to formulas by commuting with all operations. That is: where p and $A(x_1, \ldots, x_n)$ are as above and $t_1, \ldots, t_n$ are any terms, $(p(t_1, \ldots, t_n))^* = A(t_1, \ldots, t_n)$; $\bot^* = \bot$; $(\neg F)^* = \neg(F^*)$; $(F_1 \sqcap \cdots \sqcap F_k)^* = F_1^* \sqcap \cdots \sqcap F_k^*$; $(\forall x F)^* = \forall x (F^*)$; etc. For a predicate letter p, we will say "* *interprets* p *as* A'' to mean that $p^* = A$. Similarly, for a formula F, if $F^* = A$, we say that * *interprets* F *as* A.

For a formula F, an interpretation * is said to be *F-admissible* iff, for any n-ary predicate letter p, the game $A(x_1, \ldots, x_n)$ assigned to p by * does not depend on any variables that are not among $x_1, \ldots, x_n$ but occur in F. We need this condition to avoid possible collisions of variables.

Definition 5.1. A formula F is said to be *valid* iff $\vDash F^*$ for every F-admissible interpretation *.

To axiomatize the set of valid formulas, we need some technical preliminaries. Understanding $F \rightarrow G$ as an abbreviation for $\neg F \vee G$, a *positive* (resp. *negative*) *occurrence* of a subformula is one that is in the scope of an even (resp. odd) number of occurrences of $\neg$. A *surface occurrence* of a subformula is an occurrence that is not in the scope of any choice operators. A formula not containing choice operators—i.e. a formula of the classical language—is said to be *elementary*. The *elementarization* of a formula F is the result of replacing in F all surface occurrences of subformulas of the form $G_1 \sqcup \cdots \sqcup G_n$ or $\bigsqcup x G$ by $\bot$ and all surface occurrences of subformulas of the form $G_1 \sqcap \cdots \sqcap G_n$ or $\bigsqcap x G$ by $\top$. A formula is said to be *stable* iff its elementarization is classically valid. Otherwise it is *instable*.

Definition 5.2. Logic **CL3** is given by the following rules:

A. $\vec{H} \mapsto F$, where F is stable and $\vec{H}$ is a set of formulas satisfying the following conditions:
   (i) Whenever F has a positive (resp. negative) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$), for each $i \in \{1, \ldots, n\}$, $\vec{H}$ contains the result of replacing that occurrence in F by G_i;
   (ii) Whenever F has a positive (resp. negative) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$), $\vec{H}$ contains the result of replacing that occurrence in F by $G(y)$ for some variable y not occurring in F.

B1. $F' \mapsto F$, where F' is the result of replacing in F a negative (resp. positive) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) by G_i for some $i \in \{1, \ldots, n\}$.

B2. $F' \mapsto F$, where F' is the result of replacing in F a negative (resp. positive) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) by $G(t)$ for some term t such that t is not bound in F'.

Axioms are not explicitly stated, but note that the set $\vec{H}$ of premises of Rule A can be empty, in which case the conclusion F of that rule acts as an axiom. Even though this may not be immediately obvious, **CL3** essentially *is* a (refined sort of) Gentzen-style system. Consider, for example, Rule B1. It is very similar to the additive-

disjunction-introduction rule of linear logic. The only difference is that while linear logic requires that G_i be a $\vee$-(multiplicative) disjunct of the premise, **CL3** allows it to be any positive surface occurrence. This is what the calculus of structures [5] calls *deep inference*, as opposed to the *shallow inference* of linear logic. Natural semantics appear to naturally call for this sort of inference. Yet the traditional Gentzen-style axiomatizations for classical logic do not use it. To the question "why only shallow inference?'' classical logic has a simple answer: "because it is sufficient'' (for Gödel's completeness). Linear logic, however, may not have a very good answer to this or similar questions.

In the following examples and exercise, p and q are unary predicate letters.

Example 5.3. The following is a **CL3**-proof of $\sqcap x \sqcup y(p(x) \vee \neg p(y))$:
1. $p(z) \vee \neg p(z)$ (from {} by Rule A);
2. $\sqcup y(p(z) \vee \neg p(y))$ (from 1 by Rule B2);
3. $\sqcap x \sqcup y(p(x) \vee \neg p(y))$ (from {2} by Rule A).

Example 5.4. While $\exists y\, \forall x(p(x) \vee \neg p(y))$ is a classically valid elementary formula and hence derivable in **CL3** by Rule A from the empty set of premises, **CL3** does not prove its "constructive version'' $\sqcup y \sqcap x(p(x) \vee \neg p(y))$. Indeed, the latter is instable, so it could only be derived by Rule B2 from the premise $\sqcap x(p(x) \vee \neg p(t))$ for some term t different from x. Rules B1 or B2 are not applicable to $\sqcap x(p(x) \vee \neg p(t))$, so this formula could only be derived by Rule A. Then its (single) premise should be $p(z) \vee \neg p(t)$ for some variable z different from t. But $p(z) \vee \neg p(t)$ is now an instable formula not containing any choice operators, so it cannot be derived by any of the rules of **CL3**.

Exercise 5.5. With $Logic \vdash F$ (resp. $Logic \not\vdash F$) here and later meaning "F is provable (resp. not provable) in $Logic$'', verify that:
1. **CL3** $\vdash \forall x\ p(x) \rightarrow \sqcap x\ p(x)$;
2. **CL3** $\not\vdash \sqcap x\ p(x) \rightarrow \forall x\ p(x)$;
3. **CL3** proves formula (1) from Section 4;
4. **CL3** does not prove formula (2) from Section 4.

From the definition of **CL3** it is clear that if F is an elementary formula, then the only way to prove F in **CL3** is to derive it by Rule A from the empty set of premises. In particular, this rule will be applicable when F is stable, which for an elementary F means nothing but that F is classically valid. And vice versa: every classically valid formula is an elementary formula derivable in **CL3** by Rule A from the empty set of premises. Thus we have:

Proposition 5.6. *The* $\sqcap, \sqcup, \sqcap, \sqcup$*-free fragment of* **CL3** *is exactly classical logic.*

This is what we should have expected for, as noted in Section 4, when restricted to elementary problems—and elementary formulas are exactly the ones that represent such problems—the meanings of all non-choice operators of **CL3** are exactly classical.

Another natural fragment of **CL3** is the one obtained by forbidding the blind operators in its language. This is still a first-order logic as it contains the constructive quantifiers $\sqcap$ and $\sqcup$. So, the following theorem, that will be proven later in Section 10, may come as a pleasant surprise:

Theorem 5.7. *The* $\forall, \exists$*-free fragment of* **CL3** *is decidable.*

Of course **CL3** in its full language cannot be decidable as it contains classical logic. However, taking into account that classical validity and hence stability of a formula is recursively enumerable, the following fact can be immediately seen from the way **CL3** is defined:

Proposition 5.8. **CL3** *is recursively enumerable.*

Here comes our main theorem, according to which **CL3** precisely describes the set of all valid principles of computability. This theorem is just a combination of Propositions 8.1 and 9.3 proven in Part 2.

Theorem 5.9. **CL3** $\vdash F$ *iff* F *is valid* (*any formula* F). *Furthermore*:

(a) *There is an effective procedure that takes a* **CL3***-proof of a formula* F *and constructs an HPM that wins* F^* *for every* F*-admissible interpretation* *.

(b) *If* **CL3** $\not\vdash F$, *then* F^* *is not computable for some* F*-admissible interpretation* * *that interprets atoms as finitary predicates of arithmetical complexity*[6] Δ_2.

CL3 is a fragment of the logic **FD** introduced in [10]. The language of **FD** is more expressive[7] in that it has an additional sort of letters called *general letters*. Unlike our predicate letters (called *elementary letters* in [10]) that can only be interpreted as elementary games, general letters can be interpreted as any computational problems. **CL3** is obtained from **FD** by mechanically deleting the last two Rules C and D. Those two rules introduce general letters that are alien to the language we now consider. Once a general letter is introduced, it never disappears in any later formulas of an **FD**-proof. Based on this observation, a formula in our present sense is provable in **FD** iff it is provable in **CL3**, so that **FD** is a conservative extension of **CL3**. It was conjectured in [10, Conjecture 25.4] that **FD** is sound and complete with respect to computability semantics. Our Theorem 5.9 signifies a successful verification of that conjecture restricted to the general-letter-free fragment of **FD**. This fragment is called *elementary-base* as it only has elementary letters, i.e. all atoms of it represent elementary problems. The fragment of computability logic that **FD** is conjectured to axiomatize, in turn, is called *finite-depth* as all of its logical operators represent game operations that preserve the finite-depth property of games. Hence the fragment of computability logic captured by **CL3** was called in [10] the *finite-depth, elementary-base* fragment.

The language of **FD**, in turn, is just a fragment of the bigger language introduced in [10] for computability logic, called the *universal language*. The latter is the extension of the former by adding the operators $\barwedge$, $\barvee$ ($\barvee = \neg\barwedge\neg$) and $\circ\!\!-$ to it. Ref. [12] further augments the official language of computability logic with a few other natural operators. Along with the above-mentioned Conjecture 25.4 regarding the soundness and completeness of **FD**, there were two other major conjectures stated in [10] regarding the universal language: Conjectures 24.4 and 26.2. A positive verification of those two conjectures restricted to the language of **CL3** is also among the immediate consequences of our Theorem 5.9.

When restricted to the language of **CL3**, Conjecture 24.4 of [10] sounds as follows:

> *If a formula* F *is not valid, then* F^* *is not computable for some* F*-admissible interpretation* * *that interprets every atom as a finitary predicate.*[8]

The significance of this conjecture is related to the fact that showing non-validity of a given formula by appealing to interpretations that interpret atoms as infinitary predicates generally would seriously weaken such a non-validity statement. For example, if game p^* depends on infinitely many variables, then $p^* \sqcup \neg p^*$ may be incomputable just due to the fact that the machine would never be able to finish reading all the relevant information from the valuation tape necessary to determine whether p^* is true or false. On the other hand, once we restrict our considerations only to interpretations that interpret atoms as finitary predicates, the non-validity statement for $p \sqcup \neg p$ is indeed highly informative: the failure to solve $p^* \sqcup \neg p^*$ in such a case signifies fundamental limitations of algorithmic methods rather than just impossibility to obtain all the necessary external information. A positive solution to Conjecture 24.4 of [10] restricted to the language of **CL3** is contained in clause (b) of our Theorem 5.9.

As for Conjecture 26.2, it was about equivalence between validity and another version of this notion called *uniform validity*. If we disabbreviate "$\models F^*$" as "$\exists\mathcal{H}\ (\mathcal{H} \models F^*)$" (with * ranging over F-admissible interpretations and $\mathcal{H}$ over HPMs), then validity of F in the sense of Definition 5.1 can be written as "$\forall\,^*\,\exists\mathcal{H}\ (\mathcal{H} \models F^*)$". Reversing the order of quantification yields the following stronger property of uniform validity:

Definition 5.10. A formula F is said to be *uniformly valid* iff there is an HPM $\mathcal{H}$ such that, for every F-admissible interpretation *, $\mathcal{H} \models F^*$.

---

[6] See before Proposition 9.3 for an explanation of what "arithmetical complexity Δ_2" means.

[7] Ignoring the minor detail that constants were not allowed in **FD**.

[8] The original formulation of Conjecture 24.4 imposes three more restrictions on the problems through which atoms are interpreted: those problems can be chosen to also be determined (see [10] for a definition), strict (in the sense that in every position at most one of the players has legal moves) and unistructural. These conditions can be omitted in our case as they are automatically satisfied for elementary games.

Intuitively, uniform validity means existence of an interpretation-independent solution: since no information regarding interpretation * comes as a part of input to our play machines, the above HPM $\mathcal{H}$ with $\forall^*(\mathcal{H} \vDash F^*)$ will have to play in some standard, uniform way that would be successful for any possible *.

The term "uniform'' is borrowed from [1] as this understanding of validity in its spirit is close to that in Abramsky and Jagadeesan's tradition. The concepts of validity in Lorenzen's [15] tradition, or in the sense of Japaridze [8,9], also belong to this category. Common to those uniform-validity-style notions is that validity there is not defined as being "always true'' (true = ẇinnable) as this is the case with the classical understanding of this concept; in those approaches the concept of truth is often simply absent, and validity is treated as a basic concept in its own rights. As for (simply) validity, it is closer to validities in the sense of Blass [3] or Japaridze [7], and presents a direct generalization of the corresponding classical concept in that it indeed means being true (winnable) in every particular setting.

Which of our two versions of validity is more interesting depends on the motivational standpoint. It is validity rather than uniform validity that tells us what can be computed in principle. So, a computability-theoretician would focus on validity. Mathematically, non-validity is generally by an order of magnitude more informative—and correspondingly harder to prove—than non-uniform-validity. Say, the non-validity of $p \sqcup \neg p$, with the above-quoted and now successfully verified Conjecture 24.4 of [10] in mind, means existence of solvable-in-principle yet algorithmically unsolvable problems—the fact that became known to the mankind only as late as in the 20th century. As for the non-uniform-validity of $p \sqcup \neg p$, it is trivial: of course there is no way to choose one of the two disjuncts that would be true for all possible values of p because, as the Stone Age intellectuals were probably aware, some p are true and some are false.

On the other hand, it is uniform validity rather than validity that is of interest in more applied areas of computer science such as knowledgebase systems (see Section 6) or resourcebase and planning systems (see Section 26 of [10] or Section 8 of [12]). In such applications we want a logic on which a universal problem-solving machine can be based. Such a machine would or should be able to solve problems represented by formulas of **CL3** without any specific knowledge of the meaning of their atoms, i.e. without knowledge of the actual interpretation. Remembering what was said about the intuitive meaning of uniform validity, this concept is exactly what fits the bill.

Anyway, the good news, signifying a successful verification of Conjecture 26.2 of [10] restricted to the language of **CL3**, is that the two concepts of validity yield the same logic. If F is uniformly valid, then it is automatically also valid, as uniform validity is stronger than validity. Suppose now F is valid. Then, by the completeness part of Theorem 5.9, $\mathbf{CL3} \vdash F$. But then, according to the 'furthermore' clause (a) of the same theorem, F is uniformly valid. Thus, where—in accordance to our present convention—"formula'' means formula of the language of **CL3**, we have:

Theorem 5.11. *A formula is valid if and only if it is uniformly valid.*

In many contexts, such as the one of the following section, the above theorem allows us to talk about "*the* semantics'' of **CL3** without being specific regarding which of the two possible underlying concepts of validity we have in mind.

6. CL3-based applied systems

As demonstrated in Section 4, the language of **CL3** presents a convenient formalism for specifying and studying computational problems and relations between them. Its axiomatization provides a systematic way to answer not only the question '*what* can be computed' but—in view of clause (a) of Theorem 5.9—also '*how* can be computed'. Our approach brings logic and theory of computing closer together, and its general theoretical importance is obvious. The property of computability is at least as interesting as the property of (classical) truth. Moreover, as we saw, computability is also more general than truth: the latter is nothing but the former restricted to formulas of classical logic, i.e. elementary formulas. Thus, studying the logic of computability makes at least as much sense as studying the logic of truth. The latter—classical logic—is well-studied and well-explored. The former, however, has never received the treatment it naturally deserves.

The significance of our study is not limited to the theory of computation or pure logic. The fact that **CL3** is a conservative extension of classical logic makes the former a reasonable and appealing alternative to the latter in every aspect of its applications. In particular, there are good reasons to try to base applied theories—such as, say, Peano arithmetic—on **CL3** instead of just classical logic. From axioms of such a theory we would require to be "true'' in our sense, i.e. represent (under the fixed, "standard'' interpretation/model) computable problems, and from its rules of inference require to preserve the property of computability. One of the particular ways to construct such theories is

to treat the theorems of **CL3** as logical axioms and use modus ponens as the only logical rule of inference.[9] All of the non-logical axioms of the old, classical-logic-based version of the theory are true elementary formulas and hence computable in our sense, so they can be automatically included into the new set of non-logical axioms. To those could be added new, more constructive and informative axioms that involve choice operators, which would allow us to delete some or most of the old axioms that have become no longer independent. A new, computability-preserving inference rule that could be included in the **CL3**-based arithmetic is the *constructive rule of induction*:

$$\sqcap x(F(x) \rightarrow F(x+1)), \quad F(0) \mapsto \sqcap x F(x).$$

No old information whatsoever would be lost when following this path. On the other hand we would get a much more expressive, constructive and computationally meaningful theory. All theorems of such a theory would be computable problems in our sense. For example, provability of $\sqcap x \sqcup y\ p(x, y)$—as opposed to $\forall x\ \exists y\ p(x, y)$—would imply that, for every x, a y with $p(x, y)$ not only exists, but can be algorithmically found. Moreover, such an algorithm itself can be effectively constructed from a proof of the formula and algorithmic solutions (winning HPMs) to the problems represented by the non-logical axioms of the theory. This would be guaranteed by clause (a) of Theorem 5.9, Proposition 4.8 and similar facts regarding any additional, non-logical rules of inference if such are present, such as the above constructive rule of induction.

Looks like our approach materializes what the constructivists have been calling for, yet without unsettling the classically minded: whatever we could say or do by the means that classical logic offered, we can automatically still say and do, with the only difference that now many things we can say and do in a much more informative and constructive way. Our way of constructivization of theories is conservative and hence peaceful. This contrasts with many other attempts to constructivize theories, that typically try to replace classical logic by weaker logics with "constructive'' syntactic features (often in a not very clear sense)—such as intuitionistic calculus—yielding loss of information and causing the frustration of those who see nothing wrong with the classical way of reasoning.

From the purely logical point of view, it could be especially interesting to study applied theories in the $\forall$, $\exists$-free sublanguage of the language of **CL3**. Let us use **CA** to denote the version of arithmetic based on the $\forall$, $\exists$-free fragment of **CL3**. Of course, it would be more accurate to use the indefinite article "*a*'' instead of "*the*'' here, for we are not very specific about what the axioms of **CA** should be. Let us just say that **CA** has some "standard'' collection of basic axioms characterizing $=$, $+$, $\times$ and the successor function, and includes the above constructive rule of induction. For the traditional, classical-logic based version of arithmetic we use the standard name **PA**. Due to the big difference between the underlying logics of **CA** and **PA**—enough to remember that one is decidable and the other is not—**CA** might have some new and interesting features. Could we obtain a reasonably expressive yet decidable theory this way?[10] Generally, how strong a theory (whether decidable or not) could we get and what would be the fundamental limitations to its deductive power? Would **CA** still be able to numerically represent all decidable predicates and functions as **PA** does? How much of its own metatheory would **CA** be able to formalize? One can show that the property of computability of the problems represented by formulas of **CA** can be expressed in the language of **CA**, so that **CA**, unlike **PA**, would be able to talk about its own "truth''. One can also show that **PA** can constructively prove that everything provable in **CA** is true and hence **CA** is consistent. What are the limitations of the deductive strength of **CA** that make it impossible to reproduce the same proof? If there are none, then what happens to Gödel's incompleteness theorems in the context of **CA**? How about provability logic in general, which has been so well-studied for **PA** (see [13])? These are a few examples of the many intriguing questions naturally arising in this new framework and calling for answers.

CL3 can as well be of high interest in applied areas of computer science such as AI. The point is that the language of **CL3**, being a specification language for computational problems, is, at the same tame, a coherent and comprehensive query and knowledge specification language—something that the language of classical logic fails to be. Where $Age(x, y)$ is the predicate "Person x is y years old'', the knowledge represented by the formula $\forall x\ \exists y\ Age(x, y)$ is knowledge of the almost tautological fact that all people have their age. However, how to express (the stronger) knowledge of every person's actual age, which is more likely to be of relevance in a knowledgebase system? In classical logic we cannot do

[9] One could show that including some other standard logical rules such as quantification rules along with modus ponens generally would not increase the deductive power of the theory as long as non-logical axioms are (re)written in a proper manner.

[10] Even if the set of non-logical axioms of **CA** is chosen finite and the rule of induction is not included, the fact that the underlying logic is decidable does not imply the decidability of **CA** itself, for the deduction theorem for **CL3**-based theories would work in a way rather different from how it works for classical-logic-based theories.

this, and this limited expressive power precludes classical logic from serving as a satisfactory logic of knowledgebase systems, that all the time deal with the necessity to distinguish between just truth and the system's actual ability to know/find/tell what is true. Within the framework of traditional approaches, classical logic needs to be extended (say, by adding to it epistemic modalities and the like) to obtain a more or less suitable logic of knowledgebases. In our case, however, the situation is much more nice: there is no need to have separate languages and logics for *theories* on one hand and *knowledgebases* on the other hand: the same logic **CL3**, with its standard semantics, can be successfully used in both cases, without the need to modify/extend/adjust it. Back to our example, knowledge of everyone's actual age can be expressed by $\sqcap x \sqcup y\, Age(x, y)$. Obviously the ability of an agent to solve this problem means its ability to correctly tell each person's age. Within the framework of computability logic, the concept of the *knowledge* of an agent can formally be defined as the set of the queries that the agent can actually solve. The word "*query*'' here is a synonym of what we call "problem'' (game), and we prefer to use the former in this new context only because it is more common in the database and knowledgebase systems lingo.

Let us look at the query intuitions associated with our game semantics. Every formula whose main operator is $\sqcup$ or $\bigsqcup$ can be thought of as a question asked by the user (environment). For example, $Male(Dana) \sqcup Female(Dana)$ is the question "Is Dana male or female?''. Solving this problem, by our semantics for $\sqcup$, means correctly telling the gender of Dana. Formulas whose main operator is $\sqcap$ or $\bigsqcap$, on the other hand, represent questions asked by the system. For example, $\bigsqcap x(Male(x) \sqcup Female(x))$ is the question "Whose gender do you want me to tell you?''. The user's response can be "Dana'', which will bring the game down to the above user-asked question regarding the gender of Dana. Just as we noted when discussing computational problems, the language of **CL3** allows us to form queries of arbitrary complexities and degrees of interactivity. Negation turns queries into counterqueries; parallel operators generate parallel queries where both the user and the system can have simultaneous questions and counterquestions, with $\rightarrow$ acting as a query reduction operator; and blind quantifiers generate imperfect-information queries. Let us look at

$$\forall x(\bigsqcup y\, Age(x, y) \wedge (Male(x) \sqcup Female(x)) \rightarrow \bigsqcup z BestDiet(z, x)), \tag{3}$$

where $BestDiet(z, x)$ is the predicate "z is the best diet for x''. The ability of the knowledgebase system to solve this query means its ability to determine the best diet for any person, provided that the system is told that person's age and gender—that is, its ability to reduce the 'best diet' problem to the 'age and gender' problem. That x is quantified with $\forall$ rather than $\bigsqcap$ means that the system does not need to be told who the person really is. The following is a possible legal scenario of interaction over this query. The system is waiting till the user specifies, in the antecedent, the age and gender of x (without having explicitly specified the value of x). Our semantics automatically makes the system successful (winner) if the user fails to respond to either of those two counterqueries. Once responses in the antecedent are received, the system selects a diet for x. The system has been successful if the diet it selected is really the best diet for x as long as the user has told it the true age and gender of x.

Most of the real information systems are interactive, and this makes our logic, which is designed to be a logic of interactive tasks, a well-suited formal framework for them and an appealing alternative to the more traditional frameworks. Imagine a medical diagnostics system. What we would like the system to do is to tell us, for any patient x, the diagnosis y for x. That is, to solve the query *for all* x $\bigsqcup y Diagnosis(x, y)$. If here we understand '*for all*' as $\forall$, the problem has no solution: an abstract x cannot be diagnosed even by God. With '*for all*' understood as $\bigsqcap$, the query does have a solution in principle. But diagnosing a patient just based on his/her identity would require having all the relevant medical information regarding that patient, which in a real knowledgebase system is unlikely to be the case. Most likely, the query that the system solves would look like $\forall x(Q(x) \rightarrow \bigsqcup y Diagnosis(x, y))$, where $Q(x)$ is a (counter)query with questions regarding x's symptoms, blood pressure, cholesterol level, reaction to various drugs, etc. (one of such questions could be $\bigsqcup z(x = z)$, effectively turning the main quantifier $\forall x$ into $\bigsqcap x$). Most likely $Q(x)$ would not be just a $\wedge$-conjunction of such questions as this was the case with the antecedent of (3), but rather it would have a more complex structure, where what questions are asked could depend on the answers that the user gave to previous questions, yielding a long dialogue with a series of interspersed moves by both parties.

A more familiar to each of us real-life example is the automated bank account information system. You dial the bank-by-phone number to inquire about your balance. But the query that the system solves is not really as simple as $\bigsqcup x MyBalance(x)$. If this was the case, then you would be told your balance right after dialing the number. Rather, you will have to go through quite a dialogue, with all sorts of questions regarding your preferences, account type and number, secret PIN or even mother's maiden name.

The style of the above examples and the terminology employed to explain the associated intuitions are somewhat different from those that we saw in Section 4 when discussing computational problems and operations on them, or at the beginning of the present section when discussing **CL3**-based applied theories. But notice that the underlying formal semantics remains the same: whether we talk about valid principles of computability, constructive applied theories, or knowledgebase systems—in each case we deal with the same (language of) **CL3** with its standard semantics. Using the same logic **CL3** in all these cases is possible only due to Theorem 5.11 though. The reason for the failure of the principle $p \sqcup \neg p$ in the context of computability theory is that the corresponding problem may have no algorithmic solution. That is, $p \sqcup \neg p$ is not *valid*. The reason for the failure of the same principle in the context of knowledgebase systems is much simpler. An intelligent system may fail to solve the problem $Male(Dana) \sqcup \neg Male(Dana)$ not because the latter has no algorithmic solution (of course it has one), but simply because the system does not possess sufficient knowledge to determine Dana's gender. In particular, the system with empty non-logical (but perfect logical) knowledge would not be able to solve $p \sqcup \neg p$ because it is not *uniformly valid*. According to Theorem 5.11, however, validity and uniform validity are equivalent. Hence, the logic of computability, which is about what can be computed in principle, is the same as the logic of knowledgebase systems, which is about what can be actually solved by knowledge-based agents.

The point to be made here is that our approach brings together applied theories and knowledgebase systems, traditionally studied by different clans of researchers with different motivations, visions and methods. Every computability-logic-based applied theory automatically *is* a knowledgebase system, and vice versa. Knowledgebase systems can be axiomatized in exactly the same way as we would axiomatize arithmetic. The set of non-logical axioms of such a system may include atomic formulas representing factual knowledge, such as $Father(Bob,Jane)$ ("Bob is Jane's father''); it can include non-atomic elementary formulas representing general knowledge, such as $\forall x(x \times (y+1) = (x \times y) + x)$ or $\forall x(\exists y Father(x, y) \rightarrow Male(x))$; and it can include non-elementary formulas such as $\bigsqcap x \bigsqcap y \bigsqcup z(z = x \times y)$ or $\bigsqcap x \bigsqcup y Age(x, y)$, expressing the ability of the system to compute the $\times$ function or its knowledge of (ability to tell) everyone's age. These axioms would represent what can be called the *explicit knowledge* of the system—the basic set of problems/queries that the system is able to solve. And the set of theorems of such a system would represent its overall—perhaps what can be called *implicit*—knowledge. Each theorem would be a query that the system, with **CL3** built into it, is actually capable of solving: as we noted when discussing **CL3**-based applied theories, a solution to the problem/query expressed by a formula F can be automatically obtained from a proof of F and known solutions to the non-logical axioms. Furthermore, one can show that it is not even necessary for the knowledgebase system to know actual solutions (winning HPMs) for its axioms. Rather, it would suffice to have unlimited access to machines or other knowledgebase systems (*external computational/informational resources*) that solve those axioms. "Unlimited access'' here means the possibility to query (play against) those resources any finite number of times and perhaps in parallel. There is no need for the system to know how exactly those external resources do their job as long as they do it right. The system would still be able to dynamically solve any theorem F, even if no longer able to construct an actual HPM that solves F.

Extending the meaning of the term "resource'' to physical resources as well, computability-logic-based knowledgebase systems can be further generalized to resourcebase systems and systems for resource-bound planning and action. See Section 26 of [10] for a discussion and illustrations. A more elaborated discussion of applied systems based on computability logic is given in Section 8 of [12].

Part 2

This part can be considered a technical appendix to Part 1. It is exclusively devoted to proofs of our two main results: Theorem 5.9 (Sections 7–9) and Theorem 5.7 (Section 10).

7. Preliminaries

The concept of admissible interpretation can be naturally extended from formulas to sets of formulas: For a set S of formulas, an *S-admissible* interpretation is an interpretation that is F-admissible for each $F \in S$. To simplify things, we will assume throughout the rest of this paper that all the formulas we deal with are from some fixed set S, and by "interpretation'' we will always mean S-admissible interpretation.

Reiterating and extending our earlier conventions, in what follows E, F, G, H, I, J, K will be exclusively used as a metavariable for formulas, α, β for moves, *,* for interpretations, x, y, z, s, u, w for variables, a, b, c, d for constants, t for terms and e, f for valuations.

The following lemma, on which our reasoning will often rely implicitly, is just a straightforward observation:

Lemma 7.1. *For any formula $F(x_1, \ldots, x_n)$, the set $\mathbf{Lr}_e^{(F(t_1,\ldots,t_n))^*}$ does not depend on e, * or $t_1, \ldots, t_n$.*

With the above fact in mind and in accordance with our conventions from Section 2, we will usually omit the parameter e in the expression $\mathbf{Lr}_e^{F^*}$, as well as omit "with respect to e'' in the phrase "legal run of F^* with respect to e''. Remember also from Section 2 that e can as well be omitted in the expression $\mathbf{Wn}_e^A$ when A is a constant game and hence e is irrelevant.

Lemma 7.2. *Suppose x is a variable occurring in a formula F. Then, for any interpretation *, constant c and subformula G of F, $(G[x/c])^* = G^*[x/c]$.*

Proof. Assume x occurs in F. Pick an arbitrary interpretation *, constant c and subformula G of F. Our goal statement $(G[x/c])^* = G^*[x/c]$ can be proven by induction on the complexity of G. We will only outline the proof scheme. Verification of details can be done by a routine analysis of the relevant definitions, which we lazily omit and just say something like "it is easy to see that...''

Assume G is an n-ary non-logical atom $p(t_1, \ldots, t_n)$ (the case of logical atoms $\bot$, $\top$ is trivial). Let $p^* = A(x_1, \ldots, x_n)$.

First consider the case when x is not among $t_1, \ldots, t_n$. Then $G[x/c] = G$, so it would be sufficient to show that $G^* = G^*[x/c]$. But indeed, by our convention, * is F-admissible; since x occurs in F, according to the definition of F-admissible interpretation, either $A(x_1, \ldots, x_n)$ does not depend on x, or x is among $x_1, \ldots, x_n$. In either case it can be seen that $A(t_1, \ldots, t_n)$ does not depend on x. Hence $A(t_1, \ldots, t_n)[x/c] = A(t_1, \ldots, t_n)$, i.e. $G^*[x/c] = G^*$.

Next consider the case when x is among $t_1, \ldots, t_n$. For convenience of visualization, we may assume that $t_1 = \cdots = t_i = x$ and all t_j with $i < j \leqslant n$ are different from x. Then $G[x/c] = p(c, \ldots, c, t_{i+1}, \ldots, t_n)$ and hence $(G[x/c])^* = A(c, \ldots, c, t_{i+1}, \ldots, t_n)$. It is not hard to verify that $A(c, \ldots, c, t_{i+1}, \ldots, t_n) = A(t_1, \ldots, t_n)[x/c]$, so that we get $(G[x/c])^* = G^*[x/c]$. This competes our proof of the basis case of induction.

For the inductive step, let us consider the case when $G = H_1 \wedge H_2$ as an example. The following equation is based on the obvious fact that substitution of terms commutes with $\wedge$:

$$((H_1 \wedge H_2)[x/c])^* = ((H_1[x/c]) \wedge (H_2[x/c]))^*. \tag{4}$$

Next, the operation * also commutes with $\wedge$, so that we have

$$((H_1[x/c]) \wedge (H_2[x/c]))^* = (H_1[x/c])^* \wedge (H_2[x/c])^*.$$

By the induction hypothesis, $(H_1[x/c])^* = H_1^*[x/c]$ and $(H_2[x/c])^* = H_2^*[x/c]$, so we have

$$(H_1[x/c])^* \wedge (H_2[x/c])^* = (H_1^*[x/c]) \wedge (H_2^*[x/c]).$$

Since the game operation of substitution of variables obviously commutes with $\wedge$, we have

$$(H_1^*[x/c]) \wedge (H_2^*[x/c]) = (H_1^* \wedge H_2^*)[x/c].$$

Finally, again because * commutes with $\wedge$, we have

$$(H_1^* \wedge H_2^*)[x/c] = (H_1 \wedge H_2)^*[x/c]. \tag{5}$$

The chain of equations from (4) to (5) yields $((H_1 \wedge H_2)[x/c])^* = (H_1 \wedge H_2)^*[x/c]$, i.e. $(G[x/c])^* = G^*[x/c]$.

The cases with the other propositional connectives will be handled in a similar way, based on the fact that the three operations: *, $[x/c]$ (as an operation on formulas) and $[x/c]$ (as an operation on problems) commute with $\neg$, $\vee$, $\rightarrow$, $\sqcap$, $\sqcup$ just as they commute with $\wedge$. Moreover, those three operations commute with Qy as well, where Q is any of the four quantifiers and y is a variable different from x, so the case $G = QyH$ with $y \neq x$ can also be handled in a way similar to the way we handled the case $G = H_1 \wedge H_2$.

The only remaining case is $G = QxH$ (one can see that $[x/c]$ does not commute with Qx). Obviously, we have $(QxH)[x/c] = QxH$, so that

$$((QxH)[x/c])^* = (QxH)^*. \tag{6}$$

The operation * commutes with Qx, and therefore

$$(QxH)^* = Qx(H^*).$$

$Qx(H^*)$ obviously does not depend on x, which easily implies

$$Qx(H^*) = (Qx(H^*))[x/c].$$

Again by the fact that * commutes with Qx, we have

$$(Qx(H^*))[x/c] = (QxH)^*[x/c]. \tag{7}$$

The chain of equations from (6) to (7) yields $((QxH)[x/c])^* = (QxH)^*[x/c]$, i.e. $(G[x/c])^* = G^*[x/c]$. □

By a *perfect interpretation* we mean an interpretation that interprets any n-ary predicate letter p as a finitary predicate $A(x_1, \ldots, x_n)$ that does not depend on any variables others than $x_1, \ldots, x_n$. Every perfect interpretation * is nothing but a model in the classical sense (*classical model*) with domain {*constants*}—the model that interprets each constant c as the element c of the domain and interprets each n-ary predicate letter p with $p^* = A(x_1, \ldots, x_n)$ as the predicate $A(x_1, \ldots, x_n)$. Such a predicate $A(x_1, \ldots, x_n)$ is generally $\leqslant n$-ary in our sense but can be thought of as exactly n-ary under the more traditional understanding of n-ary predicates as sets of n-tuples of objects of the domain (the understanding that we slightly revised in Section 2). By a *closed* formula we mean a formula not containing free occurrences of variables.

A straightforward induction based on a routine analysis of relevant definitions reveals that:

Lemma 7.3. *For any formula F and perfect interpretation *, the game F^* (is finitary and) does not depend on any variables that do not occur free in F; hence, if F is closed, F^* is a constant game.*

With the above fact in mind and in accordance with our conventions, as long as F is closed and * is perfect, we can always safely omit the valuation parameter e in $\mathbf{Wn}_e^{F^*}$ and simply write $\mathbf{Wn}^{F^*}$ as this is done in Lemma 7.4 below.

Remembering the observations made in Section 4 about the classical behavior of our operations $\bot$, $\top$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$, we obviously have:

Lemma 7.4. *For any closed elementary formula F and perfect interpretation *, $\mathbf{Wn}^{F^*}\langle\rangle = \top$ iff F is true in * understood as a classical model.*

Based on the above fact, for a closed elementary formula F and perfect interpretation *, the phrases "F is true in *'' and "$\mathbf{Wn}^{F^*}\langle\rangle = \top$'' will be used interchangeably. Remember also from Section 2 that, for a predicate A, another way to say "$\mathbf{Wn}_e^A\langle\rangle = \top$'' or "$A$ is true at e'' is to say "$e[A]$ is true''.

Let * be an arbitrary interpretation and e an arbitrary valuation. The *perfect interpretation induced by* $(^*, e)$ is the interpretation * such that, for every n-ary predicate letter p with $p^* = A(x_1, \ldots, x_n)$, we have $p^\star = A'(x_1, \ldots, x_n)$, where $A'(x_1, \ldots, x_n)$ is the unique game such that, for any tuple $c_1, \ldots, c_n$ of constants, $A'(c_1, \ldots, c_n) = e[A(c_1, \ldots, c_n)]$. This means nothing but that $A'(x_1, \ldots, x_n)$ is the predicate such that $A'(c_1, \ldots, c_n)$ is true (at whatever valuation) iff $A(c_1, \ldots, c_n)$ is true at e. Note that while $A(c_1, \ldots, c_n)$ may depend on some hidden variables, $A'(c_1, \ldots, c_n)$ is a constant game.

For a formula F, we will be using the notation $\|F\|$ for the elementarization of F. The following two lemmas can be verified by straightforward induction on the complexity of F.

Lemma 7.5. *For any formula F, interpretation * and valuation e, $\mathbf{Wn}_e^{F^*}\langle\rangle = \mathbf{Wn}_e^{\|F\|^*}\langle\rangle$.*

Lemma 7.6. *Suppose * is the perfect interpretation induced by $(^*, e)$, and F is a closed elementary*[11] *formula. Then* $e[F^*] = F^\star$.

A valuation f is said to be *finite* iff there is a finite set $\vec{x}$ of variables such that for every variable $y \notin \vec{x}$, $f(y) = 0$. A *representation* of a finite valuation f is a set $\{x_1/c_1, \ldots, x_n/c_n\}$, where $x_1, \ldots, x_n$ are pairwise distinct variables such that each variable x with $f(x) \neq 0$ is among $x_1, \ldots, x_n$, and $c_1, \ldots, c_n$ are constants with $f(x_1) = c_1, \ldots, f(x_n) = c_n$. We will say that such a set $\{x_1/c_1, \ldots, x_n/c_n\}$ *represents* f. By abuse of terminology, we will often identify a representation of a given finite valuation with that valuation itself.

Where f is a valuation and F is a formula, fF will denote the result of substituting in F every free occurrence of every variable x by the constant $f(x)$. That is, $fF = F[x_1/f(x_1), \ldots, x_n/f(x_n)]$, where $x_1, \ldots, x_n$ are all the free variables of F. Thus, fF is always a closed formula. Generally, we say that G is an *instance* of F iff $G = fF$ for some valuation f.

We say that a valuation f is *F-distinctive* iff for any free terms t_1 and t_2 of F, as long as $t_1 \neq t_2$, we have $f(t_1) \neq f(t_2)$.

Lemma 7.7. *For any formula F, interpretation *, and valuations e and f that agree on all free variables of F, we have* $e[F^*] = e[(fF)^*]$.

Proof. Assume $F, ^*, e, f$ are as above. Let $x_1, \ldots, x_n$ be all the free variables of F, and let $c_1 = e(x_1) = f(x_1), \ldots, c_n = e(x_n) = f(x_n)$. Obviously, we have

$$e[F^*] = e[F^*[x_1/c_1, \ldots, x_n/c_n]]. \tag{8}$$

Observe that $F^*[x_1/c_1, \ldots, x_n/c_n] = (\ldots((F^*[x_1/c_1])[x_2/c_2])\ldots)[x_n/c_n]$, and similarly for "$F$'' instead of "$F^*$''. Therefore, applying Lemma 7.2 n times, we get $F^*[x_1/c_1, \ldots, x_n/c_n] = (F[x_1/c_1, \ldots, x_n/c_n])^*$ and hence

$$e[F^*[x_1/c_1, \ldots, x_n/c_n]] = e[(F[x_1/c_1, \ldots, x_n/c_n])^*]. \tag{9}$$

But $F[x_1/c_1, \ldots, x_n/c_n]$ is nothing but fF, so we have $(F[x_1/c_1, \ldots, x_n/c_n])^* = (fF)^*$ and hence

$$e[(F[x_1/c_1, \ldots, x_n/c_n])^*] = e[(fF)^*]. \tag{10}$$

Eqs. (8)–(10) yield $e[F^*] = e[(fF)^*]$. □

Now we define a function that, for a formula F and a surface occurrence O in F, returns a string α called the *F-specification of O*, which is said to *F-specify O*. In particular:

- The occurrence of F in itself is F-specified by the empty string.
- If F is $\neg G$, $\forall x G$ or $\exists x G$, then an occurrence that happens to be in G is F-specified by the same string that G-specifies that occurrence.
- If F is $G_1 \wedge \cdots \wedge G_n$, $G_1 \vee \cdots \vee G_n$ or $G_1 \rightarrow G_2$, then an occurrence that happens to be in G_i is F-specified by $i.\alpha$, where α is the G_i-specification of that occurrence.

Example: The second occurrence of $p \sqcup q$ in $F = G \vee (p \sqcup q) \vee \neg(p \rightarrow \exists x(G \wedge (p \sqcup q)))$ is F-specified by the string "3.2.2''.

With Lemma 7.2 in mind and based on Proposition 4.7, the following lemma can be easily verified by induction on the complexity of F, the routine details of which we omit:

Lemma 7.8. *For every formula F, move α and interpretation *:*

(a) $\langle\bot\alpha\rangle \in \mathbf{Lr}^{F^*}$ *iff one of the following two conditions holds*:

(1) $\alpha = \beta i$, *where β is the F-specification of a positive (resp. negative) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) and $i \in \{1, \ldots, n\}$. In this case $\langle\bot\alpha\rangle F^* = H^*$, where H is the result of substituting in F the above occurrence by G_i.*

(2) $\alpha = \beta c$, *where β is the F-specification of a positive (resp. negative) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) and $c \in \{$constants$\}$. In this case $\langle\bot\alpha\rangle F^* = H^*$, where H is the result of substituting in F the above occurrence by $G(c)$.*

[11] In fact the lemma holds for any closed formula, but for our purposes the elementary case is sufficient.

(b) $\langle\top\alpha\rangle \in \mathbf{Lr}^{F^*}$ *iff one of the following two conditions holds*:

(1) $\alpha = \beta i$, *where* β *is the F-specification of a negative* (*resp. positive*) *surface occurrence of a subformula* $G_1 \sqcap \cdots \sqcap G_n$ (*resp.* $G_1 \sqcup \cdots \sqcup G_n$) *and* $i \in \{1, \ldots, n\}$. *In this case* $\langle\top\alpha\rangle F^* = H^*$, *where H is the result of substituting in F the above occurrence by* G_i.

(2) $\alpha = \beta c$, *where* β *is the F-specification of a negative* (*resp. positive*) *surface occurrence of a subformula* $\bigsqcap x G(x)$ (*resp.* $\bigsqcup x G(x)$) *and* $c \in \{constants\}$. *In this case* $\langle\top\alpha\rangle F^* = H^*$, *where H is the result of substituting in F the above occurrence by* $G(c)$.

8. Soundness of CL3

Proposition 8.1. *If* **CL3** $\vdash F$, *then F is valid* (*any formula F*). *Moreover, there is an effective procedure that takes a* **CL3***-proof of a formula F and returns an HPM* $\mathcal{H}$ *such that, for all* *, $\mathcal{H} \vDash F^*$.

Proof. Assume **CL3** $\vdash F$. Let us fix a particular **CL3**-proof of F. We will be referring to at as "the proof'', and referring to the formulas occurring in the proof as "proof formulas''. We assume that this is a sequence (rather than tree) of formulas without repetitions, and that every proof formula comes with a fixed *justification*—a record indicating by which rule and from what premises the formula was derived.

It would be sufficient to describe an effective way of constructing an EPM $\mathcal{E}$ with 'for all *, $\mathcal{E} \vDash F^*$'. By Proposition 3.2, such an EPM $\mathcal{E}$ can then be effectively converted into an HPM $\mathcal{H}$ with 'for all *, $\mathcal{H} \vDash F^*$'.

We construct the EPM $\mathcal{E}$, that will play in the role of $\top$, as follows. At the beginning, this machine creates two records on its work tape: E to hold a formula, and f to hold (a representation of) a finite valuation. E is initialized to F, and f initialized to $\{x_1/c_1, \ldots, x_q/c_q\}$, where $x_1, \ldots, x_q$ are all the free variables of F and, for each $1 \leqslant i \leqslant q$, c_i is the value assigned to x_i by the valuation spelled on the valuation tape. After the initialization step, the machine follows the algorithm LOOP described below.

Procedure LOOP: *While E* is a proof formula, *do* one of the following, depending on which of the three rules was used (last) to derive E in the proof:

Case of Rule A: Keep granting permission until the adversary makes a move α that satisfies the conditions of one of the following two subcases, and then act as the corresponding subcase prescribes:

Subcase (i): $\alpha = \beta i$, where β E-specifies a positive (resp. negative) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) and $i \in \{1, \ldots, n\}$. Let H be the result of substituting in E the above occurrence by G_i. Then update E to H, and update f by deleting in it all pairs u/d where u is not a free variable of H.

Subcase (ii): $\alpha = \beta c$, where β E-specifies a positive (resp. negative) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) and $c \in \{constants\}$. Let H be the premise [12] of E that is the result of substituting in E the above occurrence by $G(y)$, where y does not occur in E. Then update E to H, and update f to $f \cup \{y/c\}$ (unless x did not really have free occurrences in $G(x)$, in which case f should stay the same as it was).

Case of Rule B1: Let H be the premise of E in the proof. H is the result of substituting, in E, a certain negative (resp. positive) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) by G_i for some $i \in \{1, \ldots, n\}$. Let β be the E-specification of that occurrence. Then make the move βi, update E to H, and update f by deleting in it all pairs u/d where u is not a free variable of H.

Case of Rule B2: Let H be the premise of E in the proof. H is the result of substituting, in E, a certain negative (resp. positive) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) by $G(t)$ for some term t such that t is not bound in H. Let β be the E-specification of the above occurrence of $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$). Let $c = f(t)$ if t is either a free variable of E or a constant, [13] and $c = 0$ otherwise. Then make the move βc, update E to H, and—if t is a variable—update f to $f \cup \{t/c\}$ (unless x did not really have free occurrences in $G(x)$, in which case f should stay the same as it was).

---

[12] If there are many such premises, select the lexicographically smallest one. The presence of more than one such premise, however, signifies that the proof has some (easy-to-get-rid-of) redundancies, and we may safely assume that this is not the case.

[13] Remember that when t is a constant, $f(t) = t$.

It is obvious that (the description of) $\mathcal{E}$ can be constructed effectively from the **CL3**-proof of F. What we need to do now is to show that $\mathcal{E}$ wins F^* for every *. In doing so, we will assume that $\mathcal{E}$'s adversary never makes illegal moves. By Remark 3.1, making such an assumption is perfectly legitimate.

Pick an arbitrary interpretation *, an arbitrary valuation e and an arbitrary e-computation branch B of $\mathcal{E}$. Fix Γ as the run spelled by B. Consider the work of $\mathcal{E}$ in B. For each $i \geqslant 1$ such that LOOP makes at least i iterations in B, let E_i and f_i denote the values of the records E and f at the beginning of the ith iteration of LOOP, and K_i denote $f_i E_i$. Thus, $E_1 = F$ and, by Lemma 7.7, $e[F^*] = e[K_1^*]$. Our goal is to show that B is fair and $\mathbf{Wn}_e^{F^*}\langle\Gamma\rangle = \top$, i.e. $\mathbf{Wn}_e^{K_1^*}\langle\Gamma\rangle = \top$.

Evidently E_{i+1} is always one of the premises of E_i in the proof, so that LOOP is iterated only a finite number of times. For the same reason, the value of record E is always a proof formula (incidentally, this means that the *while* condition of LOOP is always satisfied, so that the reason why LOOP is only iterated a finite number of times is simply that one of the iterations never terminates). Fix l as the number of iterations of LOOP. The lth iteration deals with the case of Rule A, for otherwise there would be a next iteration. This implies that

$$E_l \text{ is stable.} \tag{11}$$

For each i with $1 \leqslant i \leqslant l$, let Θ_i be the sequence of the moves made by the players by the beginning of the ith iteration of LOOP, where the moves made by $\mathcal{E}$ are $\top$-labeled and the moves made by its adversary $\bot$-labeled.

$$\text{For each } i \text{ with } 1 \leqslant i \leqslant l, \text{ we have } \Theta_i \in \mathbf{Lr}^{K_1^*} \text{ and } \langle\Theta_i\rangle K_1^* = K_i^*. \tag{12}$$

This statement can be proven by induction on i. The basis case with $i = 1$ is trivial. Now consider an arbitrary i with $1 \leqslant i < l$. By the induction hypothesis, $\Theta_i \in \mathbf{Lr}^{K_1^*}$ and $\langle\Theta_i\rangle K_1^* = K_i^*$. If the ith iteration of LOOP deals with the case of Rule B1 or B2, then exactly one move α is made during that iteration, and this move is by the machine, so that $\Theta_{i+1} = \langle\Theta_i, \top\alpha\rangle$. A simple analysis of the corresponding steps of our algorithm, in conjunction with Lemma 7.8(b), can convince us that $\langle\top\alpha\rangle \in \mathbf{Lr}^{K_i^*}$ and $\langle\top\alpha\rangle K_i^* = K_{i+1}^*$. With the equalities $\Theta_{i+1} = \langle\Theta_i, \top\alpha\rangle$ and $K_i^* = \langle\Theta_i\rangle K_1^*$ in mind, the former then implies $\Theta_{i+1} \in \mathbf{Lr}^{K_1^*}$ and the latter implies $\langle\Theta_{i+1}\rangle K_1^* = K_{i+1}^*$. Suppose now the ith iteration of LOOP deals with the case of Rule A. Then the machine does not make a move. This means that $\bot$ makes a move α, for otherwise we would have $i = l$. Our assumption that $\bot$ never makes illegal moves here means nothing but that $\langle\Theta_i, \bot\alpha\rangle \in \mathbf{Lr}^{K_1^*}$ and therefore (as $K_i^* = \langle\Theta_i\rangle K_1^*$) $\langle\bot\alpha\rangle \in \mathbf{Lr}^{K_i^*}$. Applying Lemma 7.8(a) to the fact that $\langle\bot\alpha\rangle \in \mathbf{Lr}^{K_i^*}$ and analyzing the corresponding steps of our algorithm, it is easy to see that $\Theta_{i+1} = \langle\Theta_i, \bot\alpha\rangle$ and $\langle\bot\alpha\rangle K_i^* = K_{i+1}^*$. Hence $\Theta_{i+1} \in \mathbf{Lr}^{K_1^*}$ and $\langle\Theta_{i+1}\rangle K_1^* = K_{i+1}^*$. Statement (12) is proven.

$$\Gamma = \Theta_l. \tag{13}$$

Indeed. Since the lth iteration of LOOP deals with the case of Rule A, $\mathcal{E}$ does not make any moves during that iteration. We claim that $\bot$ does not make any moves either, so that run Γ that is generated in the play is exactly Θ_l. To verify this claim, suppose, for a contradiction, that during the lth iteration of LOOP $\bot$ makes a move α. As we assume that $\bot$ plays legal, we should have $\langle\Theta_l, \bot\alpha\rangle \in \mathbf{Lr}^{K_1^*}$. In view of (12), this means that $\langle\bot\alpha\rangle \in \mathbf{Lr}^{K_l^*}$. From Lemma 7.8(a), just as this was observed in the proof of (12), it is obvious that then α would satisfy the conditions of either Subcase (i) or (ii), and then there would be an $(l+1)$th iteration, which, however, is not the case. Statement (13) is proven.

The fact that the last iteration of LOOP deals with the case of Rule A and $\bot$ does not make any moves during that iteration guarantees that $\mathcal{E}$ will grant permission infinitely many times during that iteration, so that branch B is fair. Thus, in order to complete our proof of Proposition 8.1, what remains to show is that $\mathbf{Wn}_e^{K_1^*}\langle\Gamma\rangle = \top$.

According to (12), Θ_l is a legal position of K_1^* and $\langle\Theta_l\rangle K_1^* = K_l^*$. This implies that $\mathbf{Wn}_e^{K_1^*}\langle\Theta_l\rangle = \mathbf{Wn}_e^{K_l^*}\langle\rangle$. But, by (13), $\Theta_l = \Gamma$. Hence

$$\mathbf{Wn}_e^{K_1^*}\langle\Gamma\rangle = \mathbf{Wn}_e^{K_l^*}\langle\rangle. \tag{14}$$

Suppose, for a contradiction, that $\mathbf{Wn}_e^{K_1^*}\langle\Gamma\rangle \neq \top$. Then, by (14), $\mathbf{Wn}_e^{K_l^*}\langle\rangle \neq \top$, whence, according to Lemma 7.5, $\mathbf{Wn}_e^{\|K_l\|^*}\langle\rangle \neq \top$. Then Lemma 7.6 implies that $\mathbf{Wn}^{\|K_l\|^\star}\langle\rangle \neq \top$, where * is the perfect interpretation induced by $(^*, e)$. That is, $\|K_l\|$ is false in * understood as a classical model. But this is impossible because, by (11), $\|E_l\|$ is classically valid and hence $\|K_l\|$, which is an instance of $\|E_l\|$, is true in all classical models. □

9. Completeness of CL3

Lemma 9.1. *Let t be any term, $F(t)$ any formula, and t' any term that does not occur in $F(t)$. Then* $\mathbf{CL3} \vdash F(t)$ *iff* $\mathbf{CL3} \vdash F(t')$.

Proof. This lemma can be proven by induction on the lengths of **CL3**-derivations. The step corresponding to Rule A will rely on a similar fact known from classical logic. The routine details of this induction are left to the reader. Our assumption that no free variable of a formula may be bound within the same formula is relevant here. □

In our completeness proof for **CL3** we will employ the complementary logic $\mathbf{CL3}'$, whose language is the same as that of **CL3** and which is given by the following rules:

A. $\vec{H} \mapsto F$, where F is instable and $\vec{H}$ is a set of formulas satisfying the following conditions:
   (i) Whenever F has a negative (resp. positive) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$), for each $i \in \{1, \ldots, n\}$, $\vec{H}$ contains the result of replacing that occurrence in F by G_i.
   (ii) Whenever F has a negative (resp. positive) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$), $\vec{H}$ contains the result of replacing that occurrence in F by $G(y)$, where y is a variable that does not occur in F.
   (iii) Whenever F has a negative (resp. positive) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) and t is a free term of F, $\vec{H}$ contains the result of replacing in F the above occurrence of $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) by $G(y)$ and [14] all free occurrences of t by y, where y is a variable that does not occur in F.

B1. $F' \mapsto F$, where F' is the result of replacing in F a positive (resp. negative) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) by G_i for some $i \in \{1, \ldots, n\}$.

B2. $F' \mapsto F$, where F' is the result of replacing in F a positive (resp. negative) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) by $G(y)$, where y is a variable that does not occur in F.

Lemma 9.2. *If* $\mathbf{CL3} \not\vdash F$, *then* $\mathbf{CL3}' \vdash F$ (*any formula F*).

Proof. We prove this lemma by induction on the complexity of F. Assume $\mathbf{CL3} \not\vdash F$. There are two cases to consider:

Case 1: F is stable. Then one of the following two subcases must hold (otherwise F would be **CL3**-derivable by Rule A):

Subcase 1.1: There is a **CL3**-unprovable formula H that is the result of replacing in F some positive (resp. negative) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) by G_i for some $i \in \{1, \ldots, n\}$. By the induction hypothesis $\mathbf{CL3}' \vdash H$, whence, by Rule B1, $\mathbf{CL3}' \vdash F$.

Subcase 1.2: There is a **CL3**-unprovable formula H that is the result of replacing in F some positive (resp. negative) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) by $G(y)$, where y is a variable that does not occur in F. By the induction hypothesis $\mathbf{CL3}' \vdash H$, whence, by Rule B2, $\mathbf{CL3}' \vdash F$.

Case 2: F is instable. Let $\vec{H}$ be a minimal set of formulas satisfying the three conditions (i)–(iii) of Rule A of $\mathbf{CL3}'$. We claim that

None of the elements of $\vec{H}$ is **CL3**-provable. (15)

To show this, consider an arbitrary element H of $\vec{H}$. One of the following three subcases must hold:

Subcase 2.1: H is the result of replacing in F a negative (resp. positive) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) by G_i for some $1 \leqslant i \leqslant n$. If $\mathbf{CL3} \vdash H$, then F would be **CL3**-derivable from H by Rule B1, which is a contradiction.

[14] "and" = "and replacing in the resulting formula".

Subcase 2.2: H is the result of replacing in F a negative (resp. positive) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) by $G(y)$ for some y not occurring in F. Just as in the previous subcase, $\mathbf{CL3} \vdash H$ is impossible, for otherwise, by Rule B2, we would have $\mathbf{CL3} \vdash F$.

Subcase 2.3: H is the result of replacing in F a negative (resp. positive) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) by $G(y)$ and all free occurrences of some term t by y, where y is a variable that does not occur in F. Notice that then $F[t/y]$ follows from H by Rule B2 of **CL3**. So, if $\mathbf{CL3} \vdash H$, then $\mathbf{CL3} \vdash F[t/y]$, and therefore, by Lemma 9.1, $\mathbf{CL3} \vdash F$. Again a contradiction, and (15) is thus proven.

Applying the induction hypothesis to (15), we conclude that each element of $\vec{H}$ is $\mathbf{CL3}'$-provable, whence, by Rule A, $\mathbf{CL3}' \vdash F$. □

Remember that a (finitary) predicate A is said to be of *complexity* Σ_2 iff it is ("can be written as'') $\exists y\, \forall z B$ for some decidable predicate B; and A is of *complexity* Δ_2 iff both A and $\neg A$ are of complexity Σ_2. The rest of this section is devoted to a proof of the following proposition:

Proposition 9.3. *If* $\mathbf{CL3} \not\vdash F$, *then* F *is not valid* (*any formula* F).

In particular, if $\mathbf{CL3} \not\vdash F$, *then* F^* *is not computable for some interpretation* * *that interprets atoms as finitary predicates of complexity* Δ_2.

Proof. Assume $\mathbf{CL3} \not\vdash F$. Then, by Lemma 9.2, $\mathbf{CL3}' \vdash F$. Let us fix a $\mathbf{CL3}'$-proof for F, call it "the proof'' and call the formulas occurring in the proof "proof formulas''. Our conventions about what a proof means are the same as in Section 8. In particular, we assume that the proof has no repetitions: every proof formula appears in it exactly once. Based on the proof, we are going to construct the fair EPM $\mathcal{E}$ which will be shown to be such that no HPM $\mathcal{H}$ wins F^* against $\mathcal{E}$ on e_c for an appropriately selected interpretation * (which does not depend on $\mathcal{H}$) and valuation e_c. Our selection of such * and e_c will be based on a diagonalization-style idea.

Let us agree for the rest of this section that $x_1, \ldots, x_q$ are all the (pairwise distinct) free variables of F, and that

Convention 9.3.1.
(a) *e always means the* (*arbitrary but fixed*) *valuation spelled on the valuation tape of* $\mathcal{E}$;
(b) *B always stands for an* (*arbitrary but fixed*) *e-computation branch of* $\mathcal{E}$.

The work of $\mathcal{E}$ consists of three stages, that we call the preinitialization, initialization and postinitialization stages. During the *preinitialization stage*, $\mathcal{E}$ checks whether e is F-distinctive (see before Lemma 7.7). If e passes the test for F-distinctiveness, $\mathcal{E}$ goes to the initialization stage. Otherwise $\mathcal{E}$ simply goes into an infinite loop in a permission state to formally ensure fairness, thus forever remaining in the preinitialization stage. During the *initialization stage*, $\mathcal{E}$ creates two records: E to hold a formula, and f to hold (a description of) a finite valuation. $\mathcal{E}$ initializes E to F and f to $\{x_1/e(x_1), \ldots, x_q/e(x_q)\}$, and goes to the postinitialization stage. During the *postinitialization stage*, $\mathcal{E}$ simply follows the following procedure:

Procedure LOOP: *While* E is a proof formula and f is an E-distinctive valuation, *do* one of the following, depending on which of the three rules was used (last) to derive E in the proof:

Case of Rule A: Keep granting permission until the adversary makes a move α. Then act depending on which of the following four subcases applies:

Subcase (i): $\alpha = \beta i$, where β E-specifies a negative (resp. positive) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) and $i \in \{1, \ldots, n\}$. Let H be the result of substituting in E the above occurrence by G_i. Then update E to H, and update f by deleting in it all pairs x/d such that x is not a free variable of H.

Subcase (ii): $\alpha = \beta c$, where β E-specifies a negative (resp. positive) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) and c is a constant not occurring in fE. Let H be the premise [15] of E that is the result of substituting in E the above occurrence by $G(y)$, where y is a variable that does not occur in E. Then

[15] As in Section 8, if there are many such premises, select the lexicographically smallest one.

update E to H, and update f to $f \cup \{y/c\}$ (unless x did not really have free occurrences in $G(x)$, in which case f should stay as it was).

Subcase (iii): $\alpha = \beta c$, where β E-specifies a negative (resp. positive) surface occurrence of a subformula $\bigsqcap xG(x)$ (resp. $\bigsqcup xG(x)$) and c is a constant that occurs in fE. Let t be the free term of E with $f(t) = c$. Let H be the premise [16] of E that is the result of substituting in E the above occurrence of $\bigsqcap xG(x)$ (resp. $\bigsqcup xG(x)$) by $G(y)$ and all free occurrences of t by y, where y is a variable that does not occur in E. Then update E to H; update f to $f \cup \{y/c\}$ if t is a constant, and to $(f - \{t/c\}) \cup \{y/c\}$ if t is a variable.

Subcase (iv): α does not satisfy any of the above conditions (i)–(iii). Then go into an infinite loop in a permission state.

Case of Rule B1: Let H be the premise of E in the proof. H is the result of substituting, in E, a certain positive (resp. negative) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) by G_i for some $i \in \{1, \ldots, n\}$. Let β be the E-specification of that occurrence. Then make the move βi, update E to H, and update f by deleting in it all pairs x/d such that x is not a free variable of H.

Case of Rule B2: Let H be the premise of E in the proof. H is the result of substituting, in E, a certain positive (resp. negative) surface occurrence of a subformula $\bigsqcap xG(x)$ (resp. $\bigsqcup xG(x)$) by $G(y)$ for some variable y not occurring in F. Let β be the E-specification of that occurrence. Let c be the smallest constant not occurring in fE. Then make the move βc, update E to H, and update f to $f \cup \{y/c\}$ (unless x did not really have free occurrences in $G(x)$, in which case f should stay as it was).

Lemma 9.3.2. *Suppose e is F-distinctive. For each $i \geqslant 1$ such that LOOP is iterated at least i times in B, let E_i and f_i be the values of E and f at the beginning of the i'th iteration. Then, for each such i (in clauses* (a)–(c)*), we have*:

(a) *E_i is a proof formula.*
(b) *f_i is an E_i-distinctive valuation.*
(c) *As long as $i > 1$, E_i is a premise of E_{i-1} in the proof.*
(d) *LOOP is iterated a finite, non-zero number of times in B.*
(e) *Where l is the number of iterations of LOOP in B, E_l is derived by Rule* A *and hence is instable.*
(f) *B is a fair branch.*

Proof. Clauses (a)–(c) are obvious from the description of LOOP. Formally they can be verified by straightforward induction on i. Note that clauses (a) and (b) imply that the *while* condition of LOOP is always satisfied.

In view of the assumption of the lemma regarding e, e will pass the test for F-distinctiveness during the preinitialization stage, so LOOP will be iterated at least once. And clause (c) implies that the number of iterations of LOOP cannot be infinite—in particular, cannot exceed the number of proof formulas. This proves clause (d).

For the remaining two clauses, assume $l \geqslant 1$ is the number of iterations of LOOP in B. As E_l is a proof formula, it should be derived by one of the three rules of **CL3′**. Among those rules, only Rule A is possible, for otherwise, as it is easy to see, we would have a next iteration of LOOP. Thus, clause (e) holds.

For clause (f), we want to show that $\mathcal{E}$ will grant permission infinitely many times—in particular, it will do so during the lth iteration of LOOP. By clause (e), the lth iteration of LOOP deals with the case of Rule A. What $\mathcal{E}$ does during that iteration is that it keeps granting permission until the adversary responds by a move. If such a response is never made, permission will be granted infinitely many times. Suppose now the adversary makes a move α. α cannot be a move that satisfies the conditions of one of the Subcases (i)–(iii), for then we would have an $(l+1)$th iteration of LOOP. Thus, we deal with Subcase (iv), in which, again, $\mathcal{E}$ will grant permission infinitely many times. □

Lemma 9.3.3. *$\mathcal{E}$ is fair.*

Proof. Keeping in mind that e and B are arbitrary (Convention 9.3.1), all we need to show is that B is fair, i.e. permission will be granted infinitely many times in B. By Lemma 9.3.2(f), if e is F-distinctive, then B is fair. And if e is not F-distinctive, then the fairness of B can be directly seen from the description of the preinitialization stage. □

[16] Again, select the lexicographically smallest one if there are many such premises.

As mentioned earlier, we are going to use $\mathcal{E}$ as an environment's strategy, so that we will be interested in runs cospelled rather than spelled by computation branches of $\mathcal{E}$. This means that when analyzing how such runs are generated, we should assume that the moves made by $\mathcal{E}$ get the label $\bot$ rather than $\top$, and the moves made by its adversary get the label $\top$ rather than $\bot$.

For the rest of this section, let us agree on the following:

Convention 9.3.4. *Suppose e is F-distinctive so that, according to Lemma* 9.3.2(d), *LOOP makes a finite, non-zero number of iterations in B. Then*:

- *l will denote the number of iterations of LOOP in B, so that the l'th iteration is the last iteration.*
- *E_i (where $1 \leqslant i \leqslant l$) will denote the value of record E at the beginning of the i'th iteration of LOOP in B.*
- *f_i (where $1 \leqslant i \leqslant l$) will denote the value of record f at the beginning of the i'th iteration of LOOP in B.*
- *K_i (where $1 \leqslant i \leqslant l$) will stand for $f_i E_i$.*
- *Θ_i (where $1 \leqslant i \leqslant l$) will stand for the sequence of the moves made by the players—in their normal order—by the beginning of the i'th iteration of LOOP in B, where the moves made by $\mathcal{E}$ are $\bot$-labeled and the moves made by its adversary $\top$-labeled.*

Lemma 9.3.5. *Suppose e is F-distinctive. Then, for every i with $1 \leqslant i \leqslant l$ and every interpretation *, we have $\Theta_i \in \mathbf{Lr}^{K_1^*}$ and $\langle \Theta_i \rangle K_1^* = K_i^*$.*

Proof. Assume e is F-distinctive. We proceed by induction on i. The basis case with $i = 1$ is trivial taking into account that $\Theta_1 = \langle\rangle$. Now consider an arbitrary i with $1 \leqslant i < l$. By the induction hypothesis, $\Theta_i \in \mathbf{Lr}^{K_1^*}$ and $\langle \Theta_i \rangle K_1^* = K_i^*$.

Suppose the ith iteration of LOOP in B deals with the case of Rule A. Then $\mathcal{E}$ does not make a move during this iteration. Since i is not the last iteration, the adversary should make a move α that satisfies the conditions of one of the Subcases (i)–(iii), and then we will have $\Theta_{i+1} = \langle \Theta_i, \top\alpha \rangle$. Analyzing how E_i and f_i are updated to E_{i+1} and f_{i+1} in this case, in view of Lemma 7.8(b) it is easy to see that then $\langle \top\alpha \rangle \in \mathbf{Lr}^{K_i^*}$ and $\langle \top\alpha \rangle K_i^* = K_{i+1}^*$, whence, with the equalities $K_i^* = \langle \Theta_i \rangle K_1^*$ and $\Theta_{i+1} = \langle \Theta_i, \top\alpha \rangle$ in mind, we have $\Theta_{i+1} \in \mathbf{Lr}^{K_1^*}$ and $\langle \Theta_{i+1} \rangle K_1^* = K_{i+1}^*$.

Suppose now the ith iteration of LOOP deals with the case of one of the Rules B1 or B2. Then the adversary does not move during this iteration. $\mathcal{E}$ makes a one single move α so that $\Theta_{i+1} = \langle \Theta_i, \bot\alpha \rangle$. Again, analyzing what kind of a move this α is and how E_i and f_i are updated to E_{i+1} and f_{i+1}, in view of Lemma 7.8(a) we can easily see that $\langle \bot\alpha \rangle \in \mathbf{Lr}^{K_i^*}$ and $\langle \bot\alpha \rangle K_i^* = K_{i+1}^*$, whence, with the equalities $K_i^* = \langle \Theta_i \rangle K_1^*$ and $\Theta_{i+1} = \langle \Theta_i, \bot\alpha \rangle$ in mind, we have $\Theta_{i+1} \in \mathbf{Lr}^{K_1^*}$ and $\langle \Theta_{i+1} \rangle K_1^* = K_{i+1}^*$. □

Lemma 9.3.6. *Suppose e is F-distinctive, and Γ is the run cospelled by B. Then, for any interpretation * with $\mathbf{Wn}_e^{K_l^*}\langle\rangle = \bot$, we have $\mathbf{Wn}_e^{F^*}\langle\Gamma\rangle = \bot$.*

Proof. Assume e is F-distinctive, B cospells Γ and $\mathbf{Wn}_e^{K_l^*}\langle\rangle = \bot$. By Lemma 9.3.5, $\Theta_l \in \mathbf{Lr}^{K_1^*}$ and $\langle \Theta_l \rangle K_1^* = K_l^*$. Since $\mathbf{Wn}_e^{K_l^*}\langle\rangle = \bot$, we then have $\mathbf{Wn}_e^{\langle \Theta_l \rangle K_1^*}\langle\rangle = \bot$, whence $\mathbf{Wn}_e^{K_1^*}\langle \Theta_l \rangle = \bot$, i.e. $\mathbf{Wn}^{e[K_1^*]}\langle \Theta_l \rangle = \bot$, i.e. $\mathbf{Wn}^{e[(f_1 E_1)^*]}\langle \Theta_l \rangle = \bot$. Then, remembering from the description of the initialization stage that f_1 agrees with e on all free variables of F and $E_1 = F$, Lemma 7.7 yields $\mathbf{Wn}^{e[F^*]}\langle \Theta_l \rangle = \bot$, i.e.

$$\mathbf{Wn}_e^{F^*}\langle \Theta_l \rangle = \bot. \tag{16}$$

Back to Γ. Obviously, Θ_l is an initial segment of Γ. Since E_l is derived by Rule A (Lemma 9.3.2(e)), the lth iteration of LOOP deals with Case of Rule A. So, $\mathcal{E}$ does not move during this iteration. If its adversary does not make moves either, then $\Theta_l = \Gamma$ and, by (16), $\mathbf{Wn}_e^{F^*}\langle\Gamma\rangle = \bot$. Suppose now the adversary makes a move α during the lth iteration. α cannot be a move that satisfies the conditions of one of the Subcases (i)–(iii), for otherwise there would be an $(l+1)$th iteration of LOOP. But if none of those three conditions is satisfied, then it can be seen from Lemma 7.8(b) that we must have $\langle \top\alpha \rangle \notin \mathbf{Lr}^{K_l^*}$. Consequently, by Lemma 9.3.5, $\langle \top\alpha \rangle \notin \mathbf{Lr}^{\langle \Theta_l \rangle K_1^*}$, whence $\langle \Theta_l, \top\alpha \rangle \notin \mathbf{Lr}^{K_1^*}$, whence, in view of Lemmas 7.1 and 7.2, $\langle \Theta_l, \top\alpha \rangle \notin \mathbf{Lr}^{F^*}$. But $\langle \Theta_l, \top\alpha \rangle$ is an initial segment of Γ, which makes Γ a $\top$-illegal and hence $\bot$-won run of $e[F^*]$. □

To proceed with our proof of Proposition 9.3, we need to agree on some additional terminology. In the following convention, when using set-theoretic notation such as $c \in \vec{c}$, we identify a tuple $\vec{c}$ of constants with the set of the constants that appear in $\vec{c}$.

Convention 9.3.7. *Suppose $\vec{a} = (a_1, \ldots, a_r)$ and $\vec{b} = (b_1, \ldots, b_r)$ are two r-tuples of pairwise distinct constants. Let $(a'_1, \ldots, a'_m)$ be the result of deleting in $\vec{a}$ all constants that are in $\vec{b}$. Similarly, let $(b'_1, \ldots, b'_m)$ be the result of deleting in $\vec{b}$ all constants that are in $\vec{a}$. We define the $(\vec{a}, \vec{b})$-permutation as the function $\hbar$: {constants} $\to$ {constants} such that, for every constant c, we have*:

- *If $c \notin (\vec{a} \cup \vec{b})$, then $\hbar c = c$.*
- *If $c = b_i \in \vec{b}$ $(1 \leqslant i \leqslant r)$, then $\hbar c = a_i$.*
- *If $c = a'_j \in (\vec{a} - \vec{b})$ $(1 \leqslant j \leqslant m)$, then $\hbar c = b'_j$.*

The following statement is obvious:

Lemma 9.3.8. *For any tuples $\vec{a} = (a_1, \ldots, a_r)$ and $\vec{b} = (b_1, \ldots, b_r)$ of pairwise distinct constants, the $(\vec{a}, \vec{b})$-permutation is an effective, bijective function from* {constants} *to* {constants}.

For the rest of this section we assume that:

Assumption 9.3.9.

- *$H_1, \ldots, H_k$ are all the instable proof formulas.*
- *$G_1(x^1_1, \ldots, x^1_{r_1}), \ldots, G_k(x^k_1, \ldots, x^k_{r_k})$ are the elementarizations of $H_1, \ldots, H_k$, respectively, where, for each $1 \leqslant i \leqslant k$, we assume that $x^i_1, \ldots, x^i_{r_i}$ are all the (pairwise distinct) free variables of $G_i(x^i_1, \ldots, x^i_{r_i})$.*

By a *Δ_2-interpretation* we mean an interpretation that interprets each predicate letter as a (finitary) predicate of complexity Δ_2.

By Gödel's completeness theorem for classical predicate calculus—in particular, the version of the proof of that theorem as given in Section 72 of [14]—for every formula $G(w_1, \ldots, w_r)$ of the classical language that is not (classically) valid and whose free variables are exactly $w_1, \ldots, w_r$, there is a classical model with domain $\{0, 1, 2, \ldots\}$ and an r-tuple $a_1, \ldots, a_r$ of pairwise distinct objects of the domain such that, in that model,

- every predicate letter is interpreted as a predicate of complexity Δ_2;
- $G(a_1, \ldots, a_r)$ is false.

Such a model is nothing but what we would call a perfect (see before Lemma 7.3) Δ_2-interpretation. Based on the above fact and taking into account that each of the $G_i(x^i_1, \ldots, x^i_{r_i})$ $(1 \leqslant i \leqslant k)$ is a classically non-valid elementary formula, we fix the following perfect Δ_2-interpretations and tuples of constants:

Assumption 9.3.10. *For each $1 \leqslant i \leqslant k$.*

- *$^{\star i}$ is a perfect Δ_2-interpretation and*
- *$\vec{a^i} = (a^i_1, \ldots, a^i_{r_i})$ are pairwise distinct constants such that $G_i(a^i_1, \ldots, a^i_{r_i})$ is false in $^{\star i}$.*

For each $1 \leqslant i \leqslant k$ and each n-ary predicate letter p, let

$$A^p_i(u_1, \ldots, u_n) = p^{\star i}$$

(of course, it is legitimate to assume that the attached tuple of each $p^{\star i}$ comes from the same pool $u_1, u_2, \ldots$ of variables).

Let us fix an effective encoding of tuples of pairwise distinct constants. We assume that every such tuple has exactly one code, and vice versa: every $c_0 \in \{0, 1, \ldots\}$ is the code of exactly one tuple of pairwise distinct constants.

For each $1 \leqslant i \leqslant k$ and each n-ary predicate letter p, we define the predicate

$$B^p_i(u_0, u_1, \ldots, u_n)$$

by stipulating that, for any $c_0, \ldots, c_n$, $B_i^p(c_0, \ldots, c_n)$ is true iff c_0 is the code of an r_i-tuple $\vec{b}$ of pairwise distinct constants and, where $\hbar$ is the $(\vec{a^i}, \vec{b})$-permutation, $A_i^p(\hbar c_1, \ldots, \hbar c_n)$ is true.

Since $\hbar$ is an effective function and the complexity of A_i^p is Δ_2, we obviously have:

Lemma 9.3.11. *For any n-ary predicate letter p and any $1 \leqslant i \leqslant k$, the complexity of $B_i^p(u_0, u_1, \ldots, u_n)$ is Δ_2.*

Remember that $x_1, \ldots, x_q$ are all the free variables of F. We also select and fix an arbitrary variable s that does not occur in F. And we fix a constant d_0 such that no constant occurring in F is greater than d_0.

For each constant c, we define the valuation e_c by stipulating that:

- $e_c(s) = c$;
- $e_c(x_1) = d_0 + 1; \quad \ldots; \quad e_c(x_q) = d_0 + q$;
- for any other variable z, $e_c(z) = 0$.

Notice that:

Lemma 9.3.12.
(a) *For any constant c, e_c is an F-distinctive valuation.*
(b) *The function g defined by $g(c, i) = e_c(v_i)$ is effective.*

We fix the list $\mathcal{H}_0, \ \mathcal{H}_1, \ \mathcal{H}_2, \ \ldots$ of all HPMs arranged according to the lexicographic order of their (standardized) descriptions.

According to Lemma 9.3.3, $\mathcal{E}$ is fair. Hence, for every HPM $\mathcal{H}$ and valuation f, the $(\mathcal{E}, f, \mathcal{H})$-branch (see Lemma 3.3) is defined.

For each constant c, we define:

- B_c as the $(\mathcal{E}, e_c, \mathcal{H}_c)$-branch,[17] and
- Γ_c as the $\mathcal{H}_c$ vs. $\mathcal{E}$ run on e_c, i.e. the run cospelled by B_c.

Note that, by Lemmas 9.3.12(a) and 9.3.2(d), LOOP is iterated a finite, non-zero number of times in B_c.

Next, where $1 \leqslant i \leqslant k$, we define the predicate $Last_i(x, x')$ by stipulating that, for any constants c, c',

- $Last_i(c, c')$ is true iff we have:
  - The value of record E in the last iteration of LOOP in B_c is H_i;
  - c' is the code of $\vec{b}$, where $\vec{b} = b_1, \ldots, b_{r_i}$ are the constants assigned to the variables $x_1^i, \ldots, x_{r_i}^i$ by the value of record f in the last iteration of LOOP in B_c. Note that, in view of Lemma 9.3.2(b), $b_1, \ldots, b_{r_i}$ are pairwise distinct.

Lemma 9.3.13. *For each $1 \leqslant i \leqslant k$, the predicate $Last_i(x, x')$ has complexity Δ_2.*

Proof. Updates of records E and f generally may take several computation steps. Let us call such steps (configurations of $\mathcal{E}$)—together with the steps within the preinitialization and initialization stages—*transitional*, and call all other steps *non-transitional*. Thus, it is the non-transitional configurations in which records E and f have definite values, with the former being a proof formula and the latter being a finite valuation. For each $1 \leqslant i \leqslant k$, let $K_i(y, x, x')$ be the predicate such that $K_i(n, c, c')$ is true iff the nth configuration of B_c is non-transitional, the value of record E in that configuration is H_i, and c' is the code of $\vec{b}$, where $\vec{b} = b_1, \ldots, b_{r_i}$ are the constants assigned to the variables $x_1^i, \ldots, x_{r_i}^i$ by the value of record f in the nth configuration. In view of Lemmas 3.3(b) and 9.3.12(b), it is not hard to see that K_i is a decidable predicate. We know that the values of records E and f should stabilize at some computation step m of B_c and never change afterwards. In particular, such an m is the first configuration of the last iteration of LOOP in B_c. With this fact in mind and some little thought, we can find that $Last_i(x, x') = \exists z \, \forall y (y \geqslant z \rightarrow K_i(y, x, x'))$ and $\neg Last_i(x, x') = \exists z \, \forall y (y \geqslant z \rightarrow \neg K_i(y, x, x'))$. This means that $Last_i(x, x')$ has complexity Δ_2. $\square$

[17] Not to confuse with the predicates B_i^p.

For any n-ary predicate letter p and any $1 \leqslant i \leqslant k$, we now define the predicate $C_i^p(s, u_1, \ldots, u_n)$ by

$$C_i^p(s, u_1, \ldots, u_n) = \exists u_0(Last_i(s, u_0) \wedge B_i^p(u_0, u_1, \ldots, u_n)).$$

For each n-ary predicate letter p, we define the predicate $D^p(u_1, \ldots, u_n)$ by

$$D^p(u_1, \ldots, u_n) = C_1^p(s, u_1, \ldots, u_n) \vee \cdots \vee C_k^p(s, u_1, \ldots, u_n).$$

(Notice that $D^p(u_1, \ldots, u_n)$ is generally $n+1$-ary rather than n-ary, with s being a hidden variable on which it depends.)

Lemma 9.3.14. *The predicate $D^p(u_1, \ldots, u_n)$ has complexity Δ_2 (any n-ary predicate letter p).*

Proof. Disjunction preserves Δ_2-complexity. So, in order to show that the predicate $D^p(u_1, \ldots, u_n)$ is of complexity Δ_2, it would be sufficient to verify that each disjunct $C_i^p(s, u_1, \ldots, u_n)$ $(1 \leqslant i \leqslant k)$ of it has complexity Δ_2. From Lemmas 9.3.11 and 9.3.13, together with the fact that $\wedge$ and $\exists$ preserve Σ_2-complexity, it is obvious that $C_i^p(s, u_1, \ldots, u_n)$ is of complexity Σ_2. Thus, what remains to show is that $\neg C_i^p(s, u_1, \ldots, u_n)$ is also of complexity Σ_2. We claim that

$$\neg C_i^p(s, u_1, \ldots, u_n) = \exists u_0(\vee\{Last_j(s, u_0) \mid j \neq i,\ j \in \{1, \ldots, k\}\} \vee (Last_i(s, u_0) \wedge \neg B_i^p(u_0, u_1, \ldots, u_n))) \tag{17}$$

($\vee S$ means the $\vee$-disjunction of the elements of S, understood as $\bot$ when S is empty). This claim can be verified based on the meanings of the predicates C_i^p and $Last_i$, and the observation that, for every (value of) s, there is exactly one $j \in \{1, \ldots, k\}$ and exactly one (value of) u_0 such that $Last_j(s, u_0)$ is true. Details of this verification are left to the reader.

Now, from Lemmas 9.3.11 and 9.3.13, together with the fact that $\wedge$, $\vee$ and $\exists$ preserve Σ_2-complexity, (17) allows us to conclude that $\neg C_i^p(s, u_1, \ldots, u_n)$ is indeed of complexity Σ_2. □

Now we define the interpretation * by stipulating that, for each n-ary predicate letter p,

$$p^* = D^p(u_1, \ldots, u_n).$$

Lemma 9.3.14 then means that * is a Δ_2-interpretation. The fact that variable s does not occur in F guarantees that this interpretation is F-admissible. What remains to show is that no HPM wins F^*. We are going to do this by proving that each $\mathcal{H}_c$ loses F^* against $\mathcal{E}$ on e_c.

Lemma 9.3.15. *Assume the following*:
(1) $c, c' \in \{\text{constants}\}$ *and* $i \in \{1, \ldots, k\}$ *are such that* $Last_i(c, c')$ *is true*;
(2) $\vec{b} = (b_1, \ldots, b_{r_i})$ *is the tuple of pairwise distinct constants encoded by* c';
(3) $\hbar$ *is the* $(\vec{a^i}, \vec{b})$*-permutation.*
Then, for any elementary formula $J(z_1, \ldots, z_n)$ *whose free variables are exactly* $z_1, \ldots, z_n$ *and any constants* $c_1, \ldots, c_n$, $e_c[(J(c_1, \ldots, c_n))^*] = (J(\hbar c_1, \ldots, \hbar c_n))^{\star i}$.

Proof. Assume the conditions of the lemma are satisfied, and $J(z_1, \ldots, z_n)$ is an elementary formula whose free variables are exactly $z_1, \ldots, z_n$. We prove the lemma by induction on the complexity of $J(z_1, \ldots, z_n)$.

For the basis of induction, we need to consider the case when $J(z_1, \ldots, z_n)$ is atomic. The cases when it is $\bot$ or $\top$ are trivial, so suppose $J(z_1, \ldots, z_n)$ is a non-logical atom. For simplicity of representation and obviously without loss of generality, we may assume that $J(z_1, \ldots, z_n) = p(z_1, \ldots, z_n)$, where p is an n-ary predicate letter. Then $(J(c_1, \ldots, c_n))^* = D^p(c_1, \ldots, c_n)$. In turn, $D^p(c_1, \ldots, c_n) = C_1^p(s, c_1, \ldots, c_n) \vee \cdots \vee C_k^p(s, c_1, \ldots, c_n)$. Thus,

$$e_c[(J(c_1, \ldots, c_n))^*] = e_c[C_1^p(s, c_1, \ldots, c_n) \vee \cdots \vee C_k^p(s, c_1, \ldots, c_n)]. \tag{18}$$

We obviously have

$$e_c[C_1^p(s, c_1, \ldots, c_n) \vee \cdots \vee C_k^p(s, c_1, \ldots, c_n)] = C_1^p(c, c_1, \ldots, c_n) \vee \cdots \vee C_k^p(c, c_1, \ldots, c_n).$$

According to assumption (1) of the lemma, $Last_i(c, c')$ is true. As noted earlier in the proof of Lemma 9.3.14, i and c' are unique values for which $Last_i(c, c')$ is true. Each component $C_j^p(c, c_1, \ldots, c_n)$ in the above disjunction contains

(under $\exists u_0$) the conjunct $Last_j(c, u_0)$ which is thus false when $j \neq i$, and hence each such disjunct $C_j^p(c, c_1, \ldots, c_n)$ can be deleted. So,

$$C_1^p(c, c_1, \ldots, c_n) \vee \cdots \vee C_k^p(c, c_1, \ldots, c_n) = C_i^p(c, c_1, \ldots, c_n) = \exists u_0(Last_i(c, u_0) \wedge B_i^p(u_0, c_1, \ldots, c_n)).$$

Since c' is the only constant for which $Last_i(c, c')$ is true, $\exists u_0(Last_i(c, u_0) \wedge B_i^p(u_0, c_1, \ldots, c_n))$ can be equivalently rewritten as $B_i^p(c', c_1, \ldots, c_n)$. Thus,

$$\exists u_0(Last_i(c, u_0) \wedge B_i^p(u_0, c_1, \ldots, c_n)) = B_i^p(c', c_1, \ldots, c_n).$$

In turn, based on assumptions (2) and (3) of the lemma,

$$B_i^p(c', c_1, \ldots, c_n) = A_i^p(\hbar c_1 \ldots, \hbar c_n).$$

Finally, notice that

$$A_i^p(\hbar c_1 \ldots, \hbar c_n) = (p(\hbar c_1 \ldots, \hbar c_n))^{\star i} = (J(\hbar c_1 \ldots, \hbar c_n))^{\star i}. \tag{19}$$

From the chain of equations from (18) to (19) we get $e_c[(J(c_1, \ldots, c_n))^*] = (J(\hbar c_1 \ldots, \hbar c_n))^{\star i}$, which completes our proof of the basis case of induction.

For the inductive step, we will only consider the case when the main operator of $J(z_1, \ldots, z_n)$ is $\exists$. The case with $\forall$ is similar, and the cases with $\neg, \wedge, \vee, \rightarrow$ are simpler or straightforward.

So, assume $J(z_1, \ldots, z_n) = \exists z_0 I(z_0, z_1, \ldots, z_n)$. Then $(J(c_1, \ldots, c_n))^* = (\exists z_0 I(z_0, c_1, \ldots, c_n))^* = \exists z_0 ((I(z_0, c_1, \ldots, c_n))^*)$ and $(J(\hbar c_1, \ldots, \hbar c_n))^{\star i} = (\exists z_0 I(z_0, \hbar c_1, \ldots, \hbar c_n))^{\star i} = \exists z_0((I(z_0, \hbar c_1, \ldots, \hbar c_n))^{\star i})$. Thus, we need to show that $e_c[\exists z_0((I(z_0, c_1, \ldots, c_n))^*)] = \exists z_0((I(z_0, \hbar c_1, \ldots, \hbar c_n))^{\star i})$. In other words, show that $e_c[\exists z_0((I(z_0, c_1, \ldots, c_n))^*)]$ is true iff $\exists z_0((I(z_0, \hbar c_1, \ldots, \hbar c_n))^{\star i})$ is so. In what follows we implicitly rely on Lemma 7.2. Suppose $e_c[\exists z_0((I(z_0, c_1, \ldots, c_n))^*)]$ is true. This means that there is a constant c_0 such that $e_c[(I(c_0, c_1, \ldots, c_n))^*]$ is true. Then, by the induction hypothesis, $(I(\hbar c_0, \hbar c_1, \ldots, \hbar c_n))^{\star i}$ is true. In turn, this implies that $\exists z_0((I(z_0, \hbar c_1, \ldots, \hbar c_n))^{\star i})$ is true. Now suppose $\exists z_0((I(z_0, \hbar c_1, \ldots, \hbar c_n))^{\star i})$ is true. This means that for some constant d, $(I(d, \hbar c_1, \ldots, \hbar c_n))^{\star i}$ is true. Since $\hbar$ is a bijection (Lemma 9.3.8), there is a constant c_0 with $\hbar c_0 = d$. Then, by the induction hypothesis, $e_c[(I(c_0, c_1, \ldots, c_n))^*]$ is true. Consequently, $e_c[\exists z_0((I(z_0, c_1, \ldots, c_n))^*)]$ is true. □

Lemma 9.3.16. *$\mathcal{H}_c$ does not win F^* against $\mathcal{E}$ on e_c (any constant c).*

Proof. Fix an arbitrary c. In what follows we rely on our Convention 9.3.1 with e_c and B_c in the roles of e and B, respectively. That is, in the present context e_c and B_c should be understood as synonyms of to what the earlier parts of the present section referred as e and B. The fact that e_c is F-distinctive (Lemma 9.3.12) allows us to use the notation established in Convention 9.3.4. According to Lemma 9.3.2(d), LOOP is iterated a finite (and non-zero) number of times—in particular, l times in B_c. Then, by clauses (a) and (e) of Lemma 9.3.2, E_l is an instable proof formula. Hence $E_l = H_i$ for one (and exactly one as we assume that the **CL3′**-proof of F has no repetitions) of the i with $1 \leqslant i \leqslant k$. Fix this i. Consider f_l—the value of record f at the beginning of the last iteration of LOOP in B_c. Let $\vec{b} = (b_1, \ldots, b_{r_i})$ be the values returned for $x_1, \ldots, x_{r_i}$ by f_l, and let c' be the code of $\vec{b}$. So, $Last_i(c, c')$ is true. Let $\hbar$ be the $(\vec{a^i}, \vec{b})$-permutation. Thus, the three conditions of Lemma 9.3.15 are satisfied. Then, according to that lemma, $e_c[(G_i(b_1, \ldots, b_{r_i}))^*] = (G_i(\hbar b_1, \ldots, \hbar b_{r_i}))^{\star i}$. But remembering the meaning of $\hbar$, we have $\hbar b_1 = a_1^i, \ldots, \hbar b_{r_i} = a_{r_i}^i$. Thus, $e_c[(G_i(b_1, \ldots, b_{r_i}))^*]$ has the same truth value as $(G_i(a_1^i, \ldots, a_{r_i}^i))^{\star i}$. According to Assumption 9.3.10, the latter is false. Then so is the former, which can be expressed by writing

$$\mathbf{Wn}_{e_c}^{(G_i(b_1,\ldots,b_{r_i}))^*}\langle\rangle = \bot. \tag{20}$$

We have $E_l = H_i$ and hence $K_l = f_l H_i$. This obviously implies that $\|K_l\| = f_l \|H_i\|$. In turn, by Assumption 9.3.9, $\|H_i\| = G_i(x_1^i, \ldots, x_{r_i}^i)$. And we also have $f_l G_i(x_1^i, \ldots, x_{r_i}^i) = G_i(b_1^i, \ldots, b_{r_i}^i)$. Thus, $\|K_l\| = G_i(b_1^i, \ldots, b_{r_i}^i)$. By (20), we then get $\mathbf{Wn}_{e_c}^{\|K_l\|^*}\langle\rangle = \bot$. This, by Lemma 7.5, implies $\mathbf{Wn}_{e_c}^{K_l^*}\langle\rangle = \bot$. Then, by Lemma 9.3.6, we get

$\mathbf{Wn}^{F^*}_{e_c}\langle\Gamma_c\rangle = \bot$. Thus Γ_c, which is the $\mathcal{H}_c$ vs. $\mathcal{E}$ run on e_c, is a lost (by $\mathcal{H}_c$) run of F^* with respect to e_c, which means that $\mathcal{H}_c$ does not win F^* against $\mathcal{E}$ on e_c. □

Lemma 9.3.16 essentially completes our proof of Proposition 9.3: that $\mathcal{H}_c$ does not win F^* against $\mathcal{E}$ on e_c clearly means that it simply does not win F^*. But every HPM is $\mathcal{H}_c$ for some c. Hence, no HPM wins F^*, and F is not valid. □

10. Decidability of the ∀,∃-free fragment of CL3

This section is devoted to a proof of Theorem 5.7. Let F be an arbitrary formula that does not contain blind quantifiers. The decidability of the question $\mathbf{CL3} \vdash F$ can be shown by induction on the complexity of F.

F is provable iff it is derivable from some provable formulas by one of the Rules A, B1, or B2. We define a procedure that tests, as described below, each of these three possibilities. If one of those three tests succeeds, the procedure returns "yes'', otherwise returns "no''.

Testing Rule A: This routine has the following three steps. The whole test is considered to have succeeded iff each of those three steps succeed.

Step 1: Check whether F is stable, i.e. whether $\|F\|$ is classically valid. Note that the latter does not contain any quantifiers. The question of classical validity of a quantifier-free formula is, of course, decidable. Thus, this step takes only a finite amount of time. If F is stable, the step has succeeded. Otherwise it has failed.

Step 2: For each positive (resp. negative) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) in F and each $1 \leqslant i \leqslant n$, see if H is provable, where H the result of replacing in F the above occurrence by G_i. Just like F, H does not contain blind quantifiers, and its complexity is lower than that of F. Hence, by the induction hypothesis, testing whether $\mathbf{CL3} \vdash H$ takes a finite amount of time. Obviously there is only a finite number of such Hs to test, so the whole Step 2 takes a finite amount of time. If all of such Hs turn out to be provable, then the step has succeeded. Otherwise it has failed.

Step 3: For each positive (resp. negative) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) in F, see if H is provable, where H the result of replacing in F the above occurrence by $G(y)$, where y is the smallest (in the lexicographic order) variable not occurring in F. As in the previous step, H is $\forall, \exists$-free and its complexity is lower than that of F, whence, by the induction hypothesis, testing whether $\mathbf{CL3} \vdash H$ takes a finite amount of time. Also, again there is only a finite number of such Hs to test, so the whole Step 3 takes a finite amount of time. If all of such Hs turn out to be provable, then the step has succeeded. Otherwise it has failed.

Before we describe how the other rules are tested, let us verify that F is derivable by Rule A iff each of the above three steps (and hence the whole test) succeeds.

Assume all three steps succeed. Success of Step 1 means that F is stable. And success of Steps 2 and 3 obviously means that there is $\vec{H}$ that satisfies conditions (i) and (ii) of Rule A. Hence F is derivable from that $\vec{H}$ by Rule A.

Now assume one of the three steps fails. We want to show that then one of the conditions of Rule A is violated for F as a possible conclusion of that rule. Indeed: failure of Step 1 means that the condition of stability of F is violated. Failure of Step 2 obviously means that there is no set $\vec{H}$ of formulas that would satisfy condition (i) of Rule A. Suppose now Step 3 fails. In particular, there is a positive (resp. negative) occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) in F such that $\mathbf{CL3} \not\vdash H$, where H is the result of replacing in F the above occurrence by $G(y)$, with y being the smallest variable not occurring in F. Let us write H as $H(y)$. In view of Lemma 9.1, for any variable y' not occurring in F, we would also have $\mathbf{CL3} \not\vdash H(y')$. This obviously means that no set $\vec{H}$ of formulas satisfies condition (ii) of Rule A.

Each of the following two routines takes a finite amount of time for the same reasons as the routines of the above-described Steps 2 and 3 did.

Testing Rule B1: For each negative (resp. positive) surface occurrence of a subformula $G_1 \sqcap \cdots \sqcap G_n$ (resp. $G_1 \sqcup \cdots \sqcup G_n$) in F and each $1 \leqslant i \leqslant n$, see if H is provable, where H the result of replacing in F the above occurrence by G_i. If one of such Hs turns out to be provable, then the test has succeeded. Otherwise it has failed. Clearly F is derivable by Rule B1 iff this test succeeds.

Testing Rule B2: For each negative (resp. positive) surface occurrence of a subformula $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) in F, do the following:

Step 1: See if H is provable, where H the result of replacing in F the above occurrence of $\bigsqcap x G(x)$ (resp. $\bigsqcup x G(x)$) by $G(y)$, where y is the smallest variable not occurring in F.

Step 2: For each free term t of F, see if H is provable, where H the result of replacing in F the above occurrence of $\sqcap xG(x)$ (resp. $\sqcup xG(x)$) by $G(t)$.

If one of the above Hs turns out to be provable, then the test has succeeded. Otherwise it has failed.

With Lemma 9.1 in mind, a little thought can convince us that F is derivable by Rule B2 iff this test succeeds.

References

[1] S. Abramsky, R. Jagadeesan, Games and full completeness for multiplicative linear logic, J. Symbolic Logic 59 (2) (1994) 543–574.

[2] J. van Benthem, Logic in Games, ILLC, University of Amsterdam, 2001 preprint.

[3] A. Blass, A game semantics for linear logic, Ann. Pure Appl. Logic 56 (1992) 183–220.

[4] J.Y. Girard, Linear logic, Theoret. Comput. Sci. 50 (1) (1987) 1–102.

[5] A. Guglielmi, L. Strassburger, Non-commutativity and MELL in the calculus of structures, Computer Science Logic, Paris, Lecture Notes in Computer Science, Vol. 2142, Springer, Berlin, 2001 pp. 54–68.

[6] J.M.E. Hyland, C.-H.L. Ong, Fair games and full completeness for multiplicative linear logic without the MIX-rule, preprint, 1993.

[7] G. Japaridze, A constructive game semantics for the language of linear logic, Ann. Pure Appl. Logic 85 (2) (1997) 87–156.

[8] G. Japaridze, The propositional logic of elementary tasks, Notre Dame J. Formal Logic 41 (2) (2000) 171–183.

[9] G. Japaridze, The logic of tasks, Ann. Pure Appl. Logic 117 (2002) 263–295.

[10] G. Japaridze, Introduction to computability logic, Ann. Pure Appl. Logic 123 (2003) 1–99.

[11] G. Japaridze, Propositional computability logic I, Trans. Comput. Logic 7 (2) (2006) 302–330.

[12] G. Japaridze, Computability logic: a formal theory of interaction, in: D. Goldin, S. Smolka, P. Wegner (Eds.), Interactive Computation: The New Paradigm, Springer, Berlin, 2006to appear.

[13] G. Japaridze, D. de Jongh, The logic of provability, in: S. Buss (Ed.), Handbook of Proof Theory, Elsevier Science B.V., North Holland, 1998, pp. 475–546.

[14] S.C. Kleene, Introduction to Metamathematics, D. van Nostrand Company, New York, Toronto, 1952.

[15] P. Lorenzen, Ein dialogisches Konstruktivitätskriterium, in: Infinitistic Methods, PWN, Proc. Symp. Foundations of Mathematics, Warsaw, 1961, pp. 193–200.